

Le logiche substrutturali

Quando pensiamo alla logica, immaginiamo regole esplicite: il modus ponens, la legge del terzo escluso, il principio di non contraddizione. Ma esistono regole molto più profonde, così fondamentali da essere quasi invisibili. Sono le **regole strutturali**, quelle che governano non tanto *cosa* possiamo inferire, ma *come* manipoliamo le premesse stesse durante il ragionamento.

Consideriamo un'inferenza elementare: da "Piove" e "Se piove, la strada è bagnata" concludiamo "La strada è bagnata". Fin qui tutto chiaro. Ma riflettiamo su cosa stiamo tacitamente assumendo:

- Possiamo aggiungere premesse inutilizzate senza invalidare il ragionamento
- Possiamo duplicare una premessa se ci serve usarla due volte
- L'ordine in cui elenchiamo le premesse è irrilevante
- Possiamo raggrupparle come vogliamo

Queste operazioni sembrano così naturali da essere trasparenti. Ma sono davvero sempre giustificate? La risposta è no, e questa consapevolezza ha dato origine a un'intera famiglia di logiche alternative: le **logiche substrutturali**.

Il termine "substrutturale" indica precisamente questo: sono logiche che rimuovono sistematicamente una o più delle regole strutturali che la logica classica dà per scontate. Il risultato è una tassonomia ricchissima di sistemi formali, ciascuno con proprietà matematiche uniche e applicazioni sorprendentemente concrete.

Parte I: Il Calcolo dei Sequenti di Gentzen

La Nascita della Proof Theory Moderna

Per capire le logiche substrutturali dobbiamo prima comprendere il loro contesto d'origine: il **calcolo dei sequenti** sviluppato da Gerhard Gentzen nel 1934. Prima di Gentzen, la logica era dominata dai sistemi assiomatici alla Hilbert: pochi assiomi, poche regole di inferenza (tipicamente solo il modus ponens), e dimostrazioni spesso tortuose e controintuitive.

Gentzen ebbe un'intuizione rivoluzionaria: invece di nascondere le manipolazioni strutturali nel funzionamento implicito del sistema, perché non renderle esplicite? Il calcolo dei sequenti fa esattamente questo.

Anatomia di un Sequente: Una Trattazione Tecnica Completa

Un **sequente** ha la forma:

$$\Gamma \vdash \Delta$$

Dove Γ (gamma) rappresenta la struttura delle premesse e Δ (delta) quella delle conclusioni. Il simbolo $\vdash$ (chiamato "turnstile") si legge "conseguo" o "si può derivare". Ma cosa sono esattamente Γ e Δ ? Questa domanda apparentemente innocua nasconde profondità sorprendenti.

Nella logica classica, Γ e Δ sono **multiset**: collezioni finite di formule dove l'ordine non conta ma le ripetizioni contano. Tecnicamente, un multiset su un insieme A è una funzione $m: A \rightarrow \mathbb{N}$ che assegna a ogni elemento una molteplicità (quante volte appare). Scriviamo $\{A, A, B\}$ per indicare che A appare due volte e B una volta. Crucialmente, $\{A, A, B\} = \{A, B, A\} = \{B, A, A\}$: l'ordine è irrilevante.

Nella logica intuizionista, Γ è ancora un multiset ma Δ contiene al massimo una formula. Questo riflette l'interpretazione costruttivista: da certe premesse posso costruire una conclusione specifica, non affermare disgiuntivamente "almeno una tra queste conclusioni vale".

Nella logica lineare, Γ e Δ sono ancora multiset ma con una semantica radicalmente diversa: ogni formula rappresenta una risorsa che deve essere usata esattamente una volta. La molteplicità conta in modo essenziale: $\{A, A\}$ significa "ho due unità della risorsa A ", non "ho la formula A che posso usare quanto voglio".

Nella logica ordinata, Γ e Δ diventano **liste** (sequenze ordinate): $[A, B, C]$ è diverso da $[B, A, C]$. L'ordine temporale o spaziale diventa significativo. Formalmente, una lista su A è definita induttivamente: la lista vuota $[]$ è una lista, e se l è una lista e $a \in A$, allora $a::l$ (a concatenato con l) è una lista.

Nella logica non-associativa, la struttura diventa ancora più ricca: Γ e Δ sono **alberi binari** completi. Non abbiamo solo A, B, C ma $(A, (B, C))$ che è diverso da $((A, B), C)$. Formalmente, definiamo induttivamente: ogni formula A è un albero (foglia), e se T_1 e T_2 sono alberi, allora (T_1, T_2) è un albero.

Le Regole Strutturali: Formalizzazione Completa

Entriamo ora nel cuore tecnico del calcolo dei sequenti. Le regole strutturali sono schemi di inferenza che manipolano la struttura di Γ e Δ senza toccare il contenuto logico delle formule.

Indebolimento (Weakening)

Indebolimento Sinistro:

$$\Gamma \vdash \Delta$$

----- (WL)

$$\Gamma, A \vdash \Delta$$

Questa regola dice: se da Γ posso derivare Δ , allora da Γ insieme ad A posso ancora derivare Δ , semplicemente ignorando A . Formalmente, se Γ è un multiset di formule e A è una formula, Γ, A denota il multiset ottenuto aggiungendo un'occorrenza di A a Γ .

Esempio concreto di applicazione: Supponiamo di aver derivato $P, Q \vdash R$. Applicando WL con $A = S$ otteniamo $P, Q, S \vdash R$. Possiamo poi applicare WL di nuovo con $B = T$ per ottenere $P, Q, S, T \vdash R$. Le formule S e T sono "inerti": non partecipano alla derivazione di R .

Indebolimento Destro:

$$\begin{array}{c} \Gamma \vdash \Delta \\ \hline \Gamma \vdash \Delta, A \end{array} \text{ (WR)}$$

Simmetricamente, se da Γ posso derivare almeno una formula in Δ , allora posso derivare almeno una tra quelle in Δ più A (potrei comunque derivare una delle formule in Δ originale).

Interpretazione computazionale: L'indebolimento corrisponde alla capacità di ignorare argomenti in una funzione. Una funzione $\lambda x. \lambda y. x$ in λ -calcolo prende due argomenti ma ignora il secondo: questo usa implicitamente l'indebolimento. Nei linguaggi con gestione esplicita delle risorse, l'indebolimento corrisponde al garbage collection automatico: posso avere un valore e semplicemente lasciarlo cadere fuori scope senza fare nulla.

Contrazione (Contraction)

Contrazione Sinistra:

$$\begin{array}{c} \Gamma, A, A \vdash \Delta \\ \hline \Gamma, A \vdash \Delta \end{array} \text{ (CL)}$$

Se posso derivare Δ avendo due copie di A tra le premesse, allora posso derivarlo avendo una sola copia. Equivalentemente: posso "duplicare" una premessa A se mi serve usarla in due posti diversi nella derivazione.

Esempio dettagliato: Consideriamo la derivazione di $(P \rightarrow Q) \rightarrow ((P \rightarrow (Q \rightarrow R)) \rightarrow (P \rightarrow R))$. Nella logica classica, questa formula è derivabile, ma richiede di usare P più volte. Specificamente:

Assumiamo $P, P \rightarrow Q, P \rightarrow (Q \rightarrow R)$

Dalla prima e seconda otteniamo Q (modus ponens)

Dalla prima e terza otteniamo $Q \rightarrow R$ (modus ponens)

Da Q e $Q \rightarrow R$ otteniamo R (modus ponens)

Qui P è usato due volte: una volta per derivare Q e una volta per derivare $Q \rightarrow R$. Questo richiede contrazione: abbiamo implicitamente "duplicato" P . Formalmente, iniziamo con $P, P, P \rightarrow Q, P \rightarrow (Q \rightarrow R) \vdash R$ e applichiamo contrazione per ottenere $P, P \rightarrow Q, P \rightarrow (Q \rightarrow R) \vdash R$.

Contrazione Destra:

$$\Gamma \vdash \Delta, A, A$$

----- (CR)

$$\Gamma \vdash \Delta, A$$

Interpretazione computazionale: La contrazione corrisponde alla possibilità di copiare valori. In linguaggi funzionali puri, ogni valore può essere copiato gratuitamente: se ho un intero x , posso usarlo in due posti diversi senza "consumarlo". Ma per risorse come file handles o puntatori unici, la copia non è gratuita o è impossibile: ecco perché linguaggi come Rust rigettano la contrazione.

Scambio (Exchange)

Scambio Sinistro:

$$\Gamma, A, B, \Pi \vdash \Delta$$

----- (EL)

$$\Gamma, B, A, \Pi \vdash \Delta$$

L'ordine in cui elenco A e B nelle premesse è irrilevante. Formalmente, se Γ e Π sono multiset (o liste in logiche ordinate), posso permutare A e B .

Nota tecnica importante: Questa regola ha senso solo se Γ è rappresentato come una struttura dove l'ordine è definito ma non significativo (multiset) o come una lista. Se Γ fosse un insieme puro (senza ripetizioni), lo scambio sarebbe automaticamente vero per la definizione di insieme. Se Γ fosse un albero binario, lo scambio potrebbe non essere formulabile direttamente.

Scambio Destro:

$$\Gamma \vdash \Delta, A, B, \Sigma$$

$$\text{----- (ER)}$$

$$\Gamma \vdash \Delta, B, A, \Sigma$$

Esempio di quando lo scambio è essenziale: Consideriamo di derivare $P, Q \vdash Q \wedge P$. Senza scambio, dovremmo mantenere P e Q nell'ordine dato. Ma la regola di introduzione della congiunzione richiede di derivare separatamente $\vdash Q$ e $\vdash P$. Se partiamo da P, Q non possiamo direttamente usare Q per la prima derivazione (è nella "posizione sbagliata"). Lo scambio ci permette di riordinare: $P, Q \vdash$ diventa $Q, P \vdash$, e ora possiamo procedere.

Le Regole Logiche: Analisi Sistemica

Ogni connettivo logico ha regole che specificano come introdurlo (quando compare nella conclusione) e come eliminarlo (quando compare nelle premesse). Questa organizzazione sistematica è uno dei grandi contributi di Gentzen.

Congiunzione ($\wedge$)

Introduzione a Destra:

$$\Gamma \vdash \Delta, A \quad \Pi \vdash \Sigma, B$$

$$\text{----- } (\wedge R)$$

$$\Gamma, \Pi \vdash \Delta, \Sigma, A \wedge B$$

Per derivare $A \wedge B$ devo poter derivare sia A che B . Crucialmente, posso usare premesse diverse (Γ vs Π) per le due derivazioni. Questo riflette che $\wedge$ è una "scelta del dimostratore": devo fornire entrambe le parti, ma posso costruirle separatamente.

Nota sulla fusione dei contesti: L'operazione Γ, Π denota l'unione dei multiset (o concatenazione delle liste, o alberi, a seconda della logica). In logica classica, $\{P, Q\} \cup \{R, S\} = \{P, Q, R, S\}$. In logica lineare, questa operazione rappresenta la combinazione delle risorse: se Γ contiene le risorse per derivare A e Π contiene quelle per derivare B , allora $\Gamma \cup \Pi$ contiene le risorse per derivare $A \wedge B$.

Eliminazione a Sinistra (due regole):

$$\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta$$

$$\text{----- } (\wedge L_1) \quad \text{----- } (\wedge L_2)$$

$$\Gamma, A \wedge B \vdash \Delta \quad \Gamma, A \wedge B \vdash \Delta$$

Se ho $A \wedge B$ tra le premesse, posso estrarre A (o B) e usarlo. Questo riflette che da una congiunzione posso ottenere ciascuno dei congiunti.

Esempio di derivazione completa:

Deriviamo $\vdash (P \wedge Q) \rightarrow P$:

$$\begin{array}{c}
 \text{----- (Id)} \\
 P \vdash P \\
 \text{----- } (\wedge L_1) \\
 P \wedge Q \vdash P \\
 \text{----- } (\rightarrow R) \\
 \vdash (P \wedge Q) \rightarrow P
 \end{array}$$

Analizziamo passo per passo:

1. Partiamo dall'assioma di identità: $P \vdash P$ (ogni formula implica se stessa)
2. Applichiamo eliminazione sinistra della congiunzione: se $P \vdash P$ allora anche $P \wedge Q \vdash P$ (possiamo estrarre P da $P \wedge Q$)
3. Applichiamo introduzione destra dell'implicazione: se $P \wedge Q \vdash P$ allora $\vdash (P \wedge Q) \rightarrow P$

Disgiunzione ($\vee$)

Introduzione a Destra (due regole):

$$\begin{array}{cc}
 \Gamma \vdash \Delta, A & \Gamma \vdash \Delta, B \\
 \text{----- } (\vee R_1) & \text{----- } (\vee R_2) \\
 \Gamma \vdash \Delta, A \vee B & \Gamma \vdash \Delta, A \vee B
 \end{array}$$

Per derivare $A \vee B$ basta derivare uno dei due (A oppure B). Questa è una "scelta dell'avversario" nella game semantics: offro una disgiunzione ma non specifico quale disgiunto è vero.

Eliminazione a Sinistra:

$$\begin{array}{c}
 \Gamma, A \vdash \Delta \quad \Pi, B \vdash \Sigma \\
 \text{----- } (\vee L)
 \end{array}$$

$$\Gamma, \Pi, A \vee B \vdash \Delta, \Sigma$$

Se ho $A \vee B$ tra le premesse, devo essere preparato per entrambi i casi: devo poter derivare la conclusione assumendo A , e devo poterla derivare assumendo B . Questa regola è più complessa delle altre perché richiede due sotto-derivazioni.

Esempio: Deriviamo $P \vee Q \vdash Q \vee P$ (commutatività):

----- (Id)	----- (Id)
$P \vdash P$	$Q \vdash Q$
----- ($\vee R_2$)	----- ($\vee R_1$)
$P \vdash Q \vee P$	$Q \vdash Q \vee P$
----- ($\vee L$)	
$P \vee Q \vdash Q \vee P$	

Leggiamo dal basso: per derivare $Q \vee P$ da $P \vee Q$, consideriamo due casi. Se P è vero (primo diisgiunto), deriviamo $Q \vee P$ scegliendo il secondo diisgiunto. Se Q è vero (secondo diisgiunto), deriviamo $Q \vee P$ scegliendo il primo diisgiunto. In entrambi i casi arriviamo alla conclusione desiderata.

Implicazione ($\rightarrow$)

Introduzione a Destra:

$$\Gamma, A \vdash \Delta, B$$

$$\text{----- } (\rightarrow R)$$

$$\Gamma \vdash \Delta, A \rightarrow B$$

Per derivare $A \rightarrow B$, assumo A come premessa aggiuntiva e derivo B . Questo corrisponde esattamente al principio di deduzione: per mostrare " A implica B ", assumo A e derivo B .

Eliminazione a Sinistra:

$$\Gamma \vdash \Delta, A \quad \Pi, B \vdash \Sigma$$

$$\text{----- } (\rightarrow L)$$

$$\Gamma, \Pi, A \rightarrow B \vdash \Delta, \Sigma$$

Se ho $A \rightarrow B$ tra le premesse, devo derivare A (primo sottoderivazione) e, assumendo B , derivare la conclusione finale (secondo sottoderivazione). Questo codifica il modus ponens: se ho $A \rightarrow B$ e ho A , ottengo B .

Nota sulla contrazione nascosta: La regola $\rightarrow L$ richiede implicitamente contrazione quando usiamo ripetutamente la stessa implicazione. Se avessimo $A \rightarrow B$, $A \rightarrow B$ e volessimo usare entrambe, dovremmo prima applicare contrazione per fondere le due occorrenze in una.

Esempio avanzato: Deriviamo il teorema di Peirce $((P \rightarrow Q) \rightarrow P) \rightarrow P$, che è valido nella logica classica ma non nell'intuizionista:

$$\begin{array}{c}
 \text{----- (Id)} \\
 P \vdash P \\
 \\
 \text{----- } (\rightarrow R) \\
 P \vdash P \rightarrow Q, P \\
 \\
 \text{----- } (\rightarrow L) \\
 P, (P \rightarrow Q) \rightarrow P \vdash P \\
 \\
 \text{----- } (\rightarrow R) \\
 (P \rightarrow Q) \rightarrow P \vdash P \\
 \\
 \text{----- } (\rightarrow R) \\
 \vdash ((P \rightarrow Q) \rightarrow P) \rightarrow P
 \end{array}$$

Questa derivazione funziona in logica classica perché possiamo avere $P \vdash P \rightarrow Q, P$: da P possiamo derivare sia P che (vacuamente) $P \rightarrow Q$. Ma nella logica intuizionista, dove il conseguente ha al più una formula, questa derivazione fallisce: non possiamo avere $P \vdash P \rightarrow Q, P$ perché il conseguente dovrebbe contenere una sola formula.

Negazione ($\neg$)

Nella logica classica, la negazione può essere definita come $\neg A := A \rightarrow \perp$ dove $\perp$ (bottom, falso) è una costante che non ha dimostrazione. Alternativamente, può avere regole proprie:

Introduzione a Destra:

$$\begin{array}{c}
 \Gamma, A \vdash \Delta \\
 \\
 \text{----- } (\neg R)
 \end{array}$$

$$\Gamma \vdash \Delta, \neg A$$

Per derivare $\neg A$, assumo A e derivo una contraddizione (rappresentata dall'insieme vuoto di conseguenti, o equivalentemente $\perp$).

Eliminazione a Sinistra:

$$\begin{array}{l} \Gamma \vdash \Delta, A \\ \hline \Gamma, \neg A \vdash \Delta \end{array} \quad (\neg L)$$

Nella logica intuizionista: La negazione è definita costruttivamente. $\neg A$ significa che da A posso derivare una contraddizione, ma non posso usare $\neg\neg A$ per ottenere A (no doppia negazione).

Nella logica lineare: La negazione diventa completamente simmetrica e involutiva. $A^\perp$ (leggi "A perp") è il duale di A , definito induttivamente:

- $(A^\perp)^\perp = A$ (involutività perfetta)
- $(A \otimes B)^\perp = A^\perp \wp B^\perp$ (De Morgan)
- $(A \wp B)^\perp = A^\perp \otimes B^\perp$
- $(A \multimap B)^\perp = A \otimes B^\perp$

Questa è una rivoluzione concettuale: la negazione non è più primitiva ma emerge dalla dualità strutturale.

Le Regole di Identità e Taglio: Il Cuore del Sistema

Assioma di Identità:

$$\begin{array}{l} \hline A \vdash A \end{array}$$

Ogni formula implica se stessa. Questo è il punto di partenza di ogni derivazione. Sembra banale ma è fondamentale: senza identità non potremmo nemmeno iniziare.

Regola del Taglio (Cut):

$$\begin{array}{l} \Gamma \vdash \Delta, A \quad \Pi, A \vdash \Sigma \\ \hline \Gamma, \Pi \vdash \Delta, \Sigma \end{array} \quad (\text{Cut})$$

$$\Gamma, \Pi \vdash \Delta, \Sigma$$

Se posso derivare A e, usando A , posso derivare qualcos'altro, allora posso derivare quel qualcos'altro direttamente senza passare per A . Questa regola codifica la transitività dell'implicazione e la composizione delle dimostrazioni.

Esempio: Supponiamo di aver derivato $P \vdash Q$ (prima derivazione) e $Q \vdash R$ (seconda derivazione). Applicando Cut con $A = Q$ otteniamo $P \vdash R$. Abbiamo "tagliato via" il lemma intermedio Q .

Perché il Cut è profondo: Il Cut permette di costruire dimostrazioni modulari. Posso dimostrare lemmi ausiliari separatamente e poi comporli. Ma introduce anche una complicazione: la formula A che compare in Cut potrebbe non essere una sottoformula di Γ , Π , Δ , o Σ . Potrebbe essere una formula arbitrariamente complessa, introdotta "creativamente" come lemma.

Il Teorema di Eliminazione del Taglio: Dimostrazione Dettagliata

Il teorema di cut-elimination è il risultato centrale di Gentzen, il "Hauptsatz" (teorema principale). Afferma:

Teorema (Gentzen, 1934): Se $\Gamma \vdash \Delta$ è derivabile nel calcolo dei sequenti con la regola del Cut, allora è derivabile anche senza usare Cut.

Conseguenze immediate:

1. **Proprietà della sottoformula:** In una derivazione senza Cut, ogni formula che appare è una sottoformula delle formule in Γ o Δ .
2. **Consistenza:** Possiamo dimostrare che la logica è consistente (non è derivabile $\vdash$ in logica intuizionista, o $\vdash \perp$ in logica classica).
3. **Decidibilità:** Per molte logiche, cut-elimination implica decidibilità del problema di derivabilità.

Sketch della dimostrazione:

La dimostrazione procede per induzione su due parametri ordinati lessicograficamente:

1. Il **grado** del Cut: la complessità della formula A che viene tagliata
2. Il **rango** del Cut: il numero totale di regole applicate nelle due sotto-derivazioni che portano al Cut

Casi base:

Caso 1: Cut su assioma. Se una delle due derivazioni è semplicemente l'assioma $A \vdash A$:

$$\begin{array}{l} \Gamma \vdash \Delta, A \quad A \vdash A \\ \hline \text{----- (Cut)} \end{array}$$

$$\Gamma \vdash \Delta, A$$

Possiamo semplicemente usare la derivazione $\Gamma \vdash \Delta, A$ direttamente, senza Cut.

Caso 2: Cut dove A non è principale. Se A appare in $\Gamma \vdash \Delta, A$ ma l'ultima regola applicata non introduce A (A era già presente prima dell'ultima regola), possiamo "spingere" il Cut verso l'alto nell'albero di derivazione.

Per esempio, se l'ultima regola era $\wedge R$:

$$\begin{array}{c}
 \Gamma \vdash \Delta, A, B \quad \Gamma \vdash \Delta, A, C \quad \Pi, A \vdash \Sigma \\
 \hline
 \Gamma \vdash \Delta, A, B \wedge C \quad \Pi, A \vdash \Sigma \\
 \hline
 \Gamma, \Pi \vdash \Delta, \Sigma, B \wedge C
 \end{array}$$

Trasformiamo in:

$$\begin{array}{c}
 \Gamma \vdash \Delta, A, B \quad \Pi, A \vdash \Sigma \quad \Gamma \vdash \Delta, A, C \quad \Pi, A \vdash \Sigma \\
 \hline
 \Gamma, \Pi \vdash \Delta, \Sigma, B \quad \Gamma, \Pi \vdash \Delta, \Sigma, C \\
 \hline
 \Gamma, \Pi \vdash \Delta, \Sigma, B \wedge C
 \end{array}$$

Ora abbiamo due Cut ma su derivazioni con rango inferiore (le derivazioni sono più corte). Per ipotesi induttiva, possiamo eliminare questi Cut.

Caso principale: Cut su formula principale

Questo è il cuore della dimostrazione. Supponiamo che A sia introdotta dall'ultima regola in entrambe le sotto-derivazioni. Per esempio, se $A = B \wedge C$:

$$\begin{array}{c}
 \Gamma \vdash \Delta, B \quad \Gamma \vdash \Delta, C \quad \Pi, B, C \vdash \Sigma \\
 \hline
 \Gamma \vdash \Delta, B \wedge C \quad \Pi, B \wedge C \vdash \Sigma
 \end{array}$$

$$\frac{}{\Gamma, \Pi \vdash \Delta, \Sigma} \text{ (Cut su } B \wedge C \text{)}$$

Trasformiamo in:

$$\frac{\frac{\Gamma \vdash \Delta, B \quad \Pi, B, C \vdash \Sigma}{\Gamma \vdash \Delta, C} \text{ (Cut su } B \text{)}}{\Gamma, \Pi, C \vdash \Delta, \Sigma} \text{ (Cut su } C \text{)}$$

$$\Gamma, \Pi, \Pi \vdash \Delta, \Sigma, \Sigma$$

Ora abbiamo due Cut ma su formule più semplici (B e C invece di $B \wedge C$). Il grado è diminuito, quindi per ipotesi induttiva possiamo eliminare questi Cut. Infine applichiamo contrazione per eliminare le duplicazioni di Π e Σ .

Nota cruciale: Questo schema funziona per tutti i connettivi, ma ogni connettivo richiede la propria analisi dei casi. Per l'implicazione è più complesso, per i quantificatori ancora di più. Ma il principio è sempre lo stesso: o il Cut è eliminabile direttamente (casi base), o possiamo trasformarlo in Cut più semplici che sono eliminabili per ipotesi induttiva.

Complessità della trasformazione: La trasformazione può causare un'esplosione nella lunghezza della derivazione. Una derivazione con n applicazioni del Cut può diventare una derivazione senza Cut di lunghezza $\text{tower}(n)$ dove $\text{tower}(n) = 2^{2^{\dots^2}}$ (torre di esponenziali alta n). Questo è inevitabile: i lemmi intermedi comprimono informazione, e eliminarli significa esplodere quella compressione.

Parte II: La Logica Lineare — La Rivoluzione delle Risorse

Genesi: Quando la Verità Non Basta

Nel 1987, il logico francese Jean-Yves Girard stava studiando la semantica computazionale della logica quando si imbatté in un problema fondamentale. La logica classica tratta le formule come **verità eterne**: se A è vera, resta vera. Posso usarla infinite volte (contrazione) o non usarla affatto (indebolimento) senza che nulla cambi.

Ma questo non corrisponde affatto alla realtà computazionale:

- Se ho **due euro**, non è la stessa cosa di averne uno solo
- Se ho **due puntatori** alla stessa struttura dati, non è la stessa cosa di averne uno
- Se **chiamo una funzione due volte**, non è la stessa cosa di chiamarla una volta
- Se **consumo una risorsa**, non posso riusarla

Girard ebbe un'intuizione radicale: e se trattassimo le formule non come verità astratte, ma come **risorse concrete** che si consumano quando le usiamo?

La Filosofia delle Risorse

L'idea centrale della **logica lineare** è semplice e rivoluzionaria: una formula A rappresenta "una unità della risorsa A ". Le conseguenze sono immediate:

- $A \otimes A$ significa "ho due unità di A ", ed è fondamentalmente diverso da A
- $A \multimap B$ significa "se consumo A , produco B "
- $!A$ significa "ho accesso illimitato ad A " (l'esponenziale)

Facciamo un esempio concreto. Consideriamo il mondo dei bar e del denaro:

- La formula € rappresenta "un euro"
- La formula Caffè rappresenta "un caffè"
- L'implicazione lineare $\text{€} \multimap \text{Caffè}$ rappresenta "posso scambiare un euro per un caffè"

Ora, con due euro posso comprare due caffè: $\text{€}, \text{€} \vdash \text{Caffè} \otimes \text{Caffè}$. Ma non è vero che $\text{€}, \text{€} \vdash \text{€}$: due euro non equivalgono logicamente a un euro! Questa non-equivalenza è precisamente ciò che la logica classica non può esprimere, perché applica automaticamente la contrazione.

Gli Otto Connettivi della Logica Lineare

La logica lineare ha una struttura molto più ricca della logica classica. Dove quest'ultima ha un solo "e" ($\wedge$) e un solo "o" ($\vee$), la logica lineare distingue tra versioni "moltiplicative" e "additive" di ogni connettivo. In totale, ci sono otto connettivi binari organizzati in quattro coppie duali.

Connettivi Moltiplicativi

Il **tensor** ($A \otimes B$) significa "ho entrambe le risorse A e B disponibili contemporaneamente". Per derivare $A \otimes B$ devo dividere il mio contesto in due parti separate: una che dimostra A e una che dimostra B . Consumo tutto.

Il **par** ($A \wp B$) è il duale del tensor: significa "devo consumare almeno una tra A e B , forse entrambe". Se ho $A \wp B$ tra le premesse, posso trattarlo come se avessi sia A che B disponibili (ma le userò in modo coordinato).

La relazione di dualità è fondamentale: $(A \otimes B)^\perp = A^\perp \wp B^\perp$, dove $A^\perp$ è la negazione lineare di A .

Connettivi Additivi

Il connettivo **with** ($A \& B$) significa "ho una risorsa che può comportarsi come A oppure come B, ma devo scegliere". Per derivare $A \& B$ devo dimostrare che posso derivare sia A che B *usando lo stesso contesto di premesse* per entrambi.

Il connettivo **plus** ($A \oplus B$) è il duale: significa "ho A oppure B, ma non so quale". È una scelta esterna, che non controllo io.

Anche qui vale la dualità: $(A \& B) \perp = A \perp \oplus B \perp$.

L'Implicazione Lineare

Il connettivo più importante è il **lollipop** ($A \multimap B$): l'implicazione lineare. Significa "se mi dai A, io ti do B, consumando A nel processo". È profondamente diversa dall'implicazione classica $A \rightarrow B$.

Nella logica classica, $A \rightarrow (B \rightarrow A)$ è un teorema (posso ignorare B). Nella logica lineare, $A \multimap (B \multimap A)$ **non è derivabile**: se consumo A per entrare nella prima implicazione, non ce l'ho più per concludere!

Le Unità

Ogni connettivo ha la sua unità:

- **1** è l'unità del tensor: rappresenta "nessuna risorsa necessaria"
- $\perp$ (bottom) è l'unità del par, duale di 1
- $\top$ (top) è l'unità del with: "qualsiasi scelta va bene"
- **0** è l'unità del plus: "nessuna scelta disponibile"

Gli Esponenziali: Controllare la Modalità

Per recuperare i comportamenti classici quando necessario, Girard introdusse due operatori modali chiamati **esponenziali**:

Il punto esclamativo **!A** (si legge "bang A" o "of course A") significa "ho accesso illimitato alla risorsa A". Se ho !A, posso:

- Usarlo una volta (dereliction)
- Non usarlo affatto (weakening)
- Duplicarlo quante volte voglio (contraction)

In altre parole, !A recupera esattamente le tre regole strutturali che la logica lineare aveva rigettato!

Il punto interrogativo **?A** (si legge "why not A") è il duale: significa "potrei aver bisogno di A un numero arbitrario di volte, incluso zero".

Tradurre la Logica Classica

Con gli esponenziali, possiamo tradurre la logica classica nella logica lineare:

- $A \wedge B$ si traduce come $!A \otimes !B$
- $A \vee B$ si traduce come $!A \oplus !B$
- $A \rightarrow B$ si traduce come $!A \multimap B$
- $\neg A$ si traduce come $!A \multimap 0$

Questo dimostra che la logica lineare è *almeno* tanto espressiva quanto la logica classica. Ma è molto più fine: distingue tra risorse che si consumano e verità che persistono.

Proof Nets: La Geometria delle Dimostrazioni

Girard non si fermò al calcolo dei sequenti. Sviluppò una rappresentazione alternativa chiamata **proof nets** (reti di dimostrazione): grafi dove i nodi sono formule e gli archi rappresentano connettivi.

Il vantaggio è enorme: due derivazioni sequenziali che differiscono solo per l'ordine di applicazione delle regole producono la stessa proof net. Queste reti catturano l'essenza geometrica della dimostrazione, eliminando la "burocrazia sintattica".

Il **criterio di Danos-Regnier** fornisce una caratterizzazione elegante di quando un grafo è una proof net valida: deve essere aciclicamente connesso rispetto a una certa operazione di "switching" sui nodi par.

Applicazioni Concrete

La logica lineare non è un gioco astratto. Ha applicazioni concrete e immediate:

Sistemi di tipo per linguaggi di programmazione: Rust usa tipi lineari (tecnicamente affini) per il suo sistema di ownership. Ogni valore ha un proprietario unico; quando trasferisci il valore, il proprietario originale non può più usarlo. Questo previene use-after-free, double-free e data races al tempo di compilazione.

Protocolli di comunicazione: Le session types usano la logica lineare per modellare protocolli. Un tipo di sessione specifica la sequenza di messaggi da inviare e ricevere. Il type checker garantisce che il protocollo sia rispettato e che non ci siano deadlock.

Quantum computing: I qubit seguono leggi lineari naturalmente. Il teorema di no-cloning dice che non puoi duplicare uno stato quantistico arbitrario: questo è esattamente il rigetto della contrazione! Linguaggi come Quipper usano tipi lineari per garantire la correttezza dei programmi quantistici.

Game semantics: La logica lineare ha un'interpretazione elegante come teoria dei giochi tra un Proponente e un Opponent. Le dimostrazioni corrispondono a strategie vincenti, e la composizione di dimostrazioni corrisponde alla composizione di strategie.

Parte III: La Logica di Rilevanza — Contro le Implicazioni Paradossali

Il Problema dei Paradossi Materiali

La logica classica definisce l'implicazione $A \rightarrow B$ come $\neg A \vee B$. Questa definizione porta a teoremi che violano profondamente l'intuizione:

Verum ex quodlibet: "Una verità implica qualsiasi cosa". Formalmente: $\vdash B \rightarrow (A \rightarrow B)$. In italiano: "Se Parigi è in Francia, allora se la Luna è di formaggio, Parigi è in Francia". Questo è un teorema della logica classica! Ma quale connessione c'è tra la geografia terrestre e ipotesi fantasiose sulla Luna?

Ex contradictione quodlibet: "Da una falsità segue qualsiasi cosa". Formalmente: $\vdash \neg A \rightarrow (A \rightarrow B)$. In altre parole: "Se la Luna non è di formaggio, allora se la Luna è di formaggio, due più due fa cinque". Di nuovo, un teorema classico, ma assurdo dal punto di vista del significato.

Questi non sono semplici curiosità tecniche. Minano la nozione stessa di cosa significhi che una cosa "implichi" un'altra. Nella logica classica, l'implicazione è **estensionale**: dipende solo dai valori di verità, non dal contenuto semantico o dalla relazione causale tra le proposizioni.

Il Principio di Rilevanza

I logici Alan Ross Anderson e Nuel Belnap, negli anni Sessanta, proposero una soluzione radicale. Introdussero un vincolo fondamentale:

PRINCIPIO DI RILEVANZA: In ogni derivazione valida di $A \rightarrow B$, la premessa A deve essere effettivamente utilizzata nella derivazione di B .

Formalmente, questo si ottiene rigettando la regola di indebolimento. Senza indebolimento, non posso introdurre premesse che poi ignoro. Ogni premessa deve contribuire attivamente alla conclusione.

Il Sistema R

Anderson e Belnap svilupparono il **sistema R** (per "Relevance") come implementazione rigorosa di questo principio.

Gli assiomi del sistema R includono:

- $A \rightarrow A$ (riflessività: ogni proposizione implica se stessa)
- $(A \rightarrow (A \rightarrow B)) \rightarrow (A \rightarrow B)$ (contrazione: se usare A due volte basta, allora basta usarlo una volta)
- Regole per congiunzione e disgiunzione che preservano la rilevanza

Crucialmente, **non sono teoremi**:

- $A \rightarrow (B \rightarrow A)$: richiederebbe di ignorare B
- $(A \rightarrow B) \rightarrow ((B \rightarrow C) \rightarrow (A \rightarrow C))$: la transitività diretta non vale!
- $(A \rightarrow (B \rightarrow C)) \rightarrow (B \rightarrow (A \rightarrow C))$: la permutazione non vale!

Queste assenze possono sembrare strane, ma riflettono il fatto che non possiamo introdurre connessioni artificiali tra proposizioni semanticamente indipendenti.

Semantica a Mondi Possibili con Relazione Ternaria

La semantica delle logiche di rilevanza è più complessa di quella classica o modale. Usa i **modelli di Routley-Meyer** con una relazione ternaria.

Un modello è una struttura che include:

- Un insieme W di "mondi" o "stati"
- Una relazione ternaria $R \subseteq W \times W \times W$
- Un'operazione $*$ che associa a ogni mondo il suo "duale"
- Una relazione di forzamento che dice quali formule sono vere in quali mondi

La condizione di verità per l'implicazione è la parte cruciale: $w \models A \rightarrow B$ se e solo se, per ogni x e y , se $R(w,x,y)$ e $x \models A$, allora $y \models B$.

Interpretazione: il mondo w "vede" che dal mondo x (dove A è vero) si può raggiungere il mondo y attraverso la relazione R , e in y deve essere vero B . La relazione R cattura la nozione di "connessione rilevante" tra premesse e conclusioni.

Applicazioni

La logica di rilevanza ha applicazioni in diverse aree:

Condizionali del linguaggio naturale: I condizionali in lingua italiana o inglese spesso richiedono rilevanza. "Se piove, le strade sono bagnate" è un condizionale rilevante (c'è connessione causale). "Se due più due fa quattro, allora Parigi è in Francia" è semanticamente assurdo, anche se materialmente vero.

Sistemi esperti: Nei sistemi di intelligenza artificiale basati su regole, vogliamo che le regole rappresentino connessioni causali genuine, non coincidenze logiche.

Database inconsistenti: Quando diverse fonti forniscono informazioni contrastanti, vogliamo tracciare le dipendenze rilevanti senza che l'intero sistema "esploda" per contraddizioni (come accadrebbe nella logica classica, dove da una contraddizione segue tutto).

Parte IV: La Logica Affine — Risorse Scaricabili

La Posizione Intermedia

La **logica affine** occupa una posizione intermedia tra la logica lineare e quella classica. Rigetta solo la contrazione, mantenendo l'indebolimento.

In termini concreti:

- **Permesso:** posso scartare premesse inutilizzate (indebolimento)
- **Vietato:** non posso duplicare premesse (no contrazione)

Filosofia: Risorse al Massimo Una Volta

In logica affine, ogni risorsa può essere usata **al massimo una volta**, ma può anche essere semplicemente scartata. Esempi concreti:

- Un **biglietto del cinema**: posso usarlo una volta per entrare, oppure non usarlo (spreendolo), ma non posso usarlo due volte.
- Una **licenza software a singola installazione**: posso installarla su una macchina, o decidere di non installarla, ma non su due macchine.
- Un **token di autenticazione usa-e-getta**: posso usarlo per una sessione o lasciarlo scadere, ma non per due sessioni.

La differenza con la logica lineare è sottile ma importante. In logica lineare pura, *devo* usare ogni risorsa esattamente una volta. In logica affine, posso decidere di non usarla (garbage collection automatico, nel gergo dei linguaggi di programmazione).

Rust: Un Caso di Studio Reale

Il linguaggio di programmazione Rust usa un sistema di tipi affine (non lineare!) per la gestione della memoria. Il sistema di **ownership** di Rust funziona così:

- Ogni valore ha un unico proprietario
- Quando il proprietario esce dallo scope, il valore viene automaticamente deallocato
- Posso trasferire il proprietà (move), ma poi il proprietario originale non può più usare il valore
- Non posso duplicare il proprietà (a meno che il tipo non implementi esplicitamente Copy)

Questo è affine, non lineare, perché posso lasciare che un valore esca dallo scope senza usarlo. È lineare nel senso che non posso usarlo due volte (no aliasing incontrollato).

Il risultato? Rust garantisce **memory safety** senza garbage collector:

- Niente use-after-free
- Niente double-free
- Niente data races nella concorrenza

Tutto questo verificato al tempo di compilazione, senza overhead a runtime. È un trionfo pratico della logica substrutturale applicata.

Separation Logic

Un'altra applicazione cruciale della logica affine è la **separation logic** di John Reynolds e Peter O'Hearn. È una logica per ragionare sulla memoria mutabile in modo compositivo.

Il connettivo chiave è la **separating conjunction** ($P * Q$): significa "P è vero in una regione di memoria e Q è vero in una regione separata". La separazione garantisce che non ci sia aliasing: le due regioni non si sovrappongono.

Questo permette di verificare proprietà di programmi imperativi in modo modulare. Se una funzione modifica solo la memoria descritta da P e un'altra solo quella descritta da Q, e $P * Q$ è vero, allora le due funzioni possono essere ragionate indipendentemente.

La separation logic è stata usata per verificare:

- Il runtime di Rust (progetto RustBelt)
- Implementazioni di strutture dati concorrenti
- Kernel di sistemi operativi
- Smart contract su blockchain

Parte V: La Logica Ordinata — Quando l'Ordine Conta

Rigettare lo Scambio

Le **logiche ordinate** (o non-commutative) rigettano la regola di scambio: l'ordine in cui elenchiamo le premesse diventa significativo. Formalmente:

$\Gamma, A, B, \Pi \vdash \Delta$

----- RIGETTATO

$\Gamma, B, A, \Pi \vdash \Delta$

Questo significa che $[A, B]$ è fondamentalmente diverso da $[B, A]$.

Il Calcolo di Lambek: Grammatica Categoriale

Il caso paradigmatico di logica ordinata è il **calcolo di Lambek**, sviluppato negli anni Cinquanta per modellare la sintassi del linguaggio naturale.

L'intuizione è semplice ma potente: le parole in una frase hanno un ordine, e quest'ordine è cruciale per il significato. "Il gatto mangia il topo" non significa la stessa cosa di "Il topo mangia il gatto".

I Tre Connettivi Fondamentali

Il calcolo di Lambek ha tre connettivi principali:

Il prodotto ($\cdot$): $A \cdot B$ significa "A immediatamente seguito da B". L'ordine è essenziale.

La divisione sinistra ($\backslash$): $A \backslash B$ significa "B dopo aver consumato A alla sinistra". Se ho un'espressione di tipo $A \backslash B$ e fornisco un A alla sua sinistra, ottengo un B.

La divisione destra ($/$): B / A significa "B dopo aver consumato A alla destra". Se ho un'espressione di tipo B / A e fornisco un A alla sua destra, ottengo un B.

Esempio: Analisi Grammaticale

Assegniamo tipi alle parole italiane:

- "Giovanni", "Maria": NP (noun phrase)
- "mangia": $(NP \backslash S) / NP$ — un verbo transitivo che prende un soggetto a sinistra e un oggetto a destra per formare una frase completa
- "dorme": $NP \backslash S$ — un verbo intransitivo che prende solo un soggetto a sinistra
- "la": NP / N — un articolo che prende un nome a destra per formare un noun phrase
- "mela": N (nome)

Ora deriviamo "Giovanni mangia la mela":

Prima combiniamo "la mela": poiché "la" è NP / N e "mela" è N, otteniamo NP.

Poi combiniamo "mangia la mela": poiché "mangia" è $(NP \backslash S) / NP$ e "la mela" è NP, otteniamo $NP \backslash S$.

Infine combiniamo "Giovanni mangia la mela": poiché "Giovanni" è NP e "mangia la mela" è $NP \backslash S$, otteniamo S (una frase completa).

Se provassimo a derivare "mangia Giovanni la mela" con gli stessi tipi, falliremmo: l'ordine sarebbe sbagliato e la derivazione non procederebbe.

Protocolli e Sequenze di Azioni

Le logiche ordinate sono naturali per modellare situazioni dove l'ordine temporale è cruciale:

Protocolli di rete: Il protocollo TCP richiede una sequenza precisa: SYN, poi SYN-ACK, poi ACK. L'ordine non può essere permutato.

Sequenze di operazioni: In un sistema di controllo industriale, "riscaldare poi mescolare" non è equivalente a "mescolare poi riscaldare".

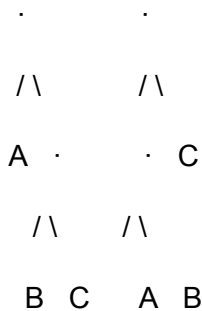
Montaggio di strutture: "Inserire la vite, poi stringere" non è equivalente a "stringere, poi inserire la vite" (e quest'ultima è impossibile).

Parte VI: La Logica Non-Associativa — Struttura Profonda

Oltre l'Ordine: Il Raggruppamento

Le logiche più radicali vanno ancora oltre, rigettando anche l'**associatività implicita**: $(A, (B, C))$ diventa strutturalmente diverso da $((A, B), C)$.

Invece di liste, la struttura diventa quella di **alberi binari**:



Questi due alberi rappresentano raggruppamenti diversi e non sono equivalenti.

Il Calcolo NL

Il **calcolo NL** (Non-associative Lambek) è la versione non-associativa del calcolo di Lambek. Le parentesi diventano obbligatorie e significative in ogni derivazione.

Applicazioni

La non-associatività è rilevante quando la struttura gerarchica conta:

Struttura costituente in linguistica: La sintassi del linguaggio naturale ha una struttura ad albero che non è arbitrariamente riassociabile. In "il gatto nero dorme", "nero" modifica "gatto", non "dorme": la struttura è $[\text{il } [\text{gatto nero}]] [\text{dorme}]$, non $[\text{il gatto}] [\text{nero dorme}]$.

Precedenza degli operatori: In matematica, $2 + 3 \times 4$ ha la struttura $2 + (3 \times 4)$, non $(2 + 3) \times 4$. Le convenzioni di precedenza codificano una struttura non-associativa.

Parsing di espressioni: In informatica, $f(g(x, y), h(z))$ ha una struttura ad albero precisa che deve essere preservata durante l'interpretazione.

Parte VII: La Gerarchia Completa e le Interrelazioni

Il Cubo delle Logiche Substructurali

Possiamo immaginare tutte queste logiche come vertici di uno spazio tridimensionale, un cubo dove ogni asse rappresenta la presenza o assenza di una regola strutturale. Il primo asse indica se ammettiamo o rigettiamo l'indebolimento, il secondo riguarda la contrazione, il

terzo lo scambio. Ogni combinazione di scelte genera una logica diversa con proprietà matematiche uniche.

Al vertice dove tutte e tre le regole sono presenti troviamo la logica classica, il sistema che conosciamo dalle elementari. È il punto di massima permissività: possiamo duplicare, scartare e riordinare le premesse a piacimento. Spostandoci lungo l'asse dell'indebolimento verso la sua assenza, mantenendo contrazione e scambio, arriviamo alla logica di rilevanza: un sistema dove ogni premessa deve essere genuinamente usata, dove non possiamo più introdurre ipotesi irrilevanti solo perché ci fa comodo.

Se invece rigettiamo la contrazione ma manteniamo indebolimento e scambio, otteniamo la logica affine. Qui le risorse non possono essere duplicate ma possono essere scartate, come il sistema di ownership di Rust: posso decidere di non usare un valore, ma se lo uso posso farlo una sola volta. Eliminando anche lo scambio, mantenendo solo l'indebolimento, arriviamo a una logica ordinata affine dove l'ordine conta ma possiamo ancora scartare risorse.

La logica lineare pura si trova nel vertice opposto a quello classico: nessuna delle tre regole strutturali è ammessa. È il punto di massima restrittività, dove ogni risorsa va usata esattamente una volta, nell'ordine in cui appare, senza possibilità di duplicazione o scarto. Da qui, aggiungendo solo lo scambio, arriviamo alla logica lineare commutativa che Girard studiò originariamente. Aggiungendo invece la contrazione ma non lo scambio otteniamo una logica ordinata con duplicazione, utile per modellare sequenze ripetibili ma dove l'ordine temporale resta cruciale.

Proprietà Matematiche e Complessità Computazionale

Ogni logica in questo spazio ha proprietà matematiche distinte che ne determinano non solo l'espressività teorica ma anche la difficoltà pratica di lavorarci. La complessità computazionale del problema di derivabilità varia drammaticamente a seconda delle regole strutturali ammesse.

La logica classica proposizionale ha un problema di derivabilità che è coNP-completo: verificare se una formula è un teorema è difficile quanto verificare che non lo sia, e appartiene alla classe dei problemi intrattabili ma non impossibili. Sorprendentemente, la logica intuizionista proposizionale è più difficile: il suo problema di derivabilità è PSPACE-completo, richiedendo potenzialmente una quantità di memoria polinomiale nella dimensione del problema. Questo è controintuitivo perché l'intuizionismo è spesso visto come più restrittivo, eppure decidere cosa sia derivabile è computazionalmente più oneroso.

La logica lineare proposizionale completa ha la stessa complessità della logica intuizionista: PSPACE-completo. Tuttavia, se ci limitiamo al frammento moltiplicativo della logica lineare, eliminando i connettivi additivi e gli esponenziali, la complessità scende drasticamente a NP-completo. Questo è un risultato notevole: rigettare le regole strutturali può effettivamente rendere il ragionamento più efficiente, almeno per certi frammenti del linguaggio.

Il calcolo di Lambek, essenzialmente una logica lineare ordinata, è anch'esso NP-completo. L'assenza dello scambio non aumenta la complessità, anzi in certi casi la semplifica perché riduce lo spazio delle possibili derivazioni da esplorare. La versione non-associativa, il

calcolo NL, mantiene la stessa complessità: NP-completo. In questi sistemi, il problema di decidere se una sequenza di parole ha una certa struttura grammaticale può essere risolto in tempo non-deterministico polinomiale, il che è sorprendentemente efficiente dato quanto siano espressive queste logiche.

Quando introduciamo quantificatori, però, la situazione cambia radicalmente. La logica lineare del secondo ordine, con quantificatori su tipi, diventa indecidibile: non esiste alcun algoritmo che possa sempre determinare se una formula è derivabile. Questo rispecchia un pattern generale: l'aggiunta di quantificazione illimitata tende a portare all'indcidibilità, indipendentemente dalle regole strutturali sottostanti.

Il Teorema di Cut-Elimination: Cuore Pulsante della Teoria

Abbiamo menzionato il teorema di cut-elimination come risultato fondamentale di Gentzen, ma vale la pena approfondirne la portata e le implicazioni. Questo teorema non è solo un risultato tecnico: è il cuore pulsante di tutta la teoria della dimostrazione moderna, e la sua validità o invalidità per una data logica ci dice moltissimo sulla natura computazionale di quel sistema.

La regola del Cut permette di introdurre lemmi intermedi in una dimostrazione: se posso derivare A e, assumendo A, posso derivare B, allora posso derivare B direttamente. Superficialmente sembra solo un principio di modularità, un modo per strutturare dimostrazioni complesse in pezzi più gestibili. Ma è molto di più. Il Cut permette di introdurre formule che non appaiono né nelle premesse né nella conclusione, formule "creative" che emergono dall'interno della dimostrazione stessa.

Il teorema di cut-elimination afferma che questa creatività è illusoria: tutto ciò che può essere dimostrato usando lemmi intermedi può essere dimostrato anche procedendo per pura analisi delle formule date. Ogni dimostrazione con Cut può essere trasformata in una dimostrazione equivalente senza Cut, dove ogni formula che appare è una sottoformula delle premesse o della conclusione.

La dimostrazione di questo teorema procede per induzione, ma la trasformazione può causare un'esplosione nella lunghezza della derivazione. Una dimostrazione con pochi Cut ben scelti può diventare, una volta eliminati i Cut, esponenzialmente più lunga. Questo non è solo un problema pratico: ci dice qualcosa di profondo sulla natura della compressione informativa nelle dimostrazioni. I lemmi intermedi sono una forma di compressione: ci permettono di non ripetere sottodimostrazioni identiche, di dare nomi a concetti complessi e riusarli. Eliminarli significa esplodere questa compressione.

Eppure, nonostante questa possibile esplosione, il teorema di cut-elimination è preziosissimo. Garantisce che il calcolo dei sequenti gode della proprietà della sottoformula, che a sua volta implica la decidibilità per molte logiche. Se ogni formula in una derivazione è una sottoformula della conclusione, allora esiste solo un numero finito di formule che potrebbero apparire in una dimostrazione di quella conclusione. Questo limita drasticamente lo spazio di ricerca.

Per le logiche substructurali, il teorema di cut-elimination è valido con alcune notevoli eccezioni. La logica lineare gode di cut-elimination, e questo è stato uno dei risultati chiave

di Girard: nonostante la complessità aggiunta dai connettivi moltiplicativi, additivi ed esponenziali, il sistema è ancora analitico. La logica affine, mantenendo l'indebolimento, preserva questa proprietà. Il calcolo di Lambek e il suo fratello non-associativo NL godono entrambi di cut-elimination, il che è cruciale per le loro applicazioni linguistiche: garantisce che il parsing di frasi sia decidibile e praticabile.

La logica di rilevanza è più problematica. Per il sistema R base, il cut-elimination è stato dimostrato ma la prova è estremamente complessa e richiede macchinari tecnici sofisticati. Per alcuni sistemi di rilevanza più forti, la questione è rimasta aperta per decenni. L'assenza di cut-elimination non significa che la logica sia incoerente o inutilizzabile, ma indica che manca quella trasparenza analitica che rende altre logiche così trattabili.

Teoremi di Interpolazione e Separazione

Un altro modo per misurare la "buona natura" di una logica è verificare se soddisfa il teorema di interpolazione di Craig. Questo teorema afferma qualcosa di apparentemente semplice ma profondamente significativo: se A implica logicamente B, allora esiste una formula intermedia C, espressa solo nel linguaggio comune ad A e B, tale che A implica C e C implica B.

L'interpolante C cattura esattamente l'informazione condivisa tra A e B, niente di più e niente di meno. La sua esistenza garantisce una forma di modularità: posso analizzare l'implicazione tra A e B scomponendola in due implicazioni più semplici attraverso la loro intersezione concettuale.

La logica classica soddisfa l'interpolazione, e così fa la logica intuizionista. La logica lineare moltiplicativa, il frammento più puro e restrittivo, gode anch'essa di questa proprietà. Ma quando aggiungiamo gli esponenziali, la situazione si complica: la logica lineare completa non ha interpolazione in generale. Questo è significativo: gli esponenziali, pur essendo necessari per recuperare la potenza espressiva della logica classica, introducono una complessità che rompe la modularità garantita dall'interpolazione.

Il calcolo di Lambek, sorprendentemente, soddisfa l'interpolazione. Questo ha conseguenze pratiche per la grammatica categoriale: significa che possiamo analizzare strutture sintattiche complesse scomponendole in sottostrutture modulari. Per la logica di rilevanza il quadro è misto: alcuni sistemi godono di interpolazione, altri no, e per alcuni la questione è ancora aperta.

I teoremi di separazione vanno nella direzione opposta: invece di cercare punti comuni tra formule, chiedono se connettivi diversi siano genuinamente distinti o se uno possa essere definito in termini degli altri. Nella logica lineare, tutti gli otto connettivi sono mutuamente separabili: nessuno può essere definito come combinazione degli altri. Questo dimostra che la struttura della logica lineare è irriducibile, non contiene ridondanze nascoste. Ogni connettivo cattura un aspetto genuinamente diverso della gestione delle risorse.

Parte VIII: Semantica e Interpretazioni dei Mondi

Algebre Residueate: L'Unificazione Semantica

Per comprendere veramente le logiche substructurali non basta studiare la sintassi delle derivazioni. Dobbiamo anche esplorare i loro mondi semantici, le strutture matematiche che danno significato alle formule. E qui scopriamo qualcosa di meraviglioso: esiste una struttura algebrica unificante che cattura l'essenza di quasi tutte le logiche substructurali. Si chiamano algebre residueate.

Un'algebra residueata consiste in un insieme dotato di diverse operazioni che interagiscono tra loro in modi precisi. Innanzitutto abbiamo una struttura di reticolo, che cattura le operazioni di congiunzione e disgiunzione: un ordinamento parziale dove ogni coppia di elementi ha un minimo superiore (congiunzione) e un massimo inferiore (disgiunzione). Poi abbiamo un'operazione di prodotto, che corrisponde alla combinazione di risorse o alla giustapposizione di premesse. Questa operazione forma un monoide: è associativa e ha un elemento neutro, ma sostanzialmente non è necessariamente commutativa né idempotente.

La magia sta nella relazione tra il prodotto e l'implicazione, catturata dalla condizione di residueazione: $a \leq b$ è minore o uguale a c se e solo se a è minore o uguale a " b implica c ". In altre parole, l'implicazione è il residuo del prodotto, quello che manca per passare da b a c quando abbiamo già a . Questa condizione definisce completamente l'implicazione in termini del prodotto e dell'ordine.

Nelle algebre residueate non-commutative, dove il prodotto dipende dall'ordine, abbiamo addirittura due implicazioni: l'implicazione destra e quella sinistra. L'implicazione destra " a implica-destra b " rappresenta ciò che, aggiunto a destra di a , produce b . L'implicazione sinistra rappresenta ciò che, aggiunto a sinistra, produce b . Questo corrisponde esattamente alle due divisioni del calcolo di Lambek.

Consideriamo un esempio concreto di algebra residueata: i linguaggi formali su un alfabeto. L'insieme è la collezione di tutti i linguaggi (insiemi di stringhe). Il prodotto è la concatenazione di linguaggi: se L_1 e L_2 sono linguaggi, il loro prodotto è l'insieme di tutte le stringhe ottenute concatenando una stringa di L_1 con una di L_2 . Questo prodotto non è commutativo: concatenare "gatto" con "nero" non dà lo stesso risultato di concatenare "nero" con "gatto".

L'implicazione destra L_1 implica-destra L_2 è l'insieme di tutte le stringhe w tali che, per ogni stringa u in L_1 , la concatenazione uw appartiene a L_2 . In altre parole, è l'insieme delle stringhe che, aggiunte a destra di qualsiasi stringa di L_1 , producono una stringa di L_2 . Similmente per l'implicazione sinistra, ma con la concatenazione a sinistra. Questa costruzione cattura esattamente la semantica del calcolo di Lambek per linguaggi formali.

Frame di Kripke Estesi: Mondi e Accessibilità

La semantica a mondi possibili, sviluppata originariamente per le logiche modali, si estende in modi affascinanti alle logiche substructurali. L'idea di base è semplice: invece di valutare le formule come semplicemente vere o false, le valutiamo relativamente a "mondi" o "stati" o "punti". Una formula può essere vera in un mondo ma falsa in un altro.

Nella logica modale standard, abbiamo una relazione binaria di accessibilità tra mondi: un mondo w può "vedere" un mondo v . La formula "necessariamente A " è vera in w se A è vera in tutti i mondi che w può vedere. Questo framework si è rivelato straordinariamente fertile per interpretare nozioni come necessità, conoscenza, credenza, tempo e obbligazione.

Per le logiche substructurali, però, una relazione binaria non basta. Abbiamo bisogno di relazioni ternarie e persino quaternarie. Nella logica lineare, una relazione ternaria $R(w,x,y)$ significa che il mondo w "vede" che i mondi x e y possono essere combinati.

L'interpretazione di $A \otimes B$ in w è che esistono mondi x e y tali che $R(w,x,y)$, A è vero in x , e B è vero in y . In altre parole, w riconosce una decomposizione delle risorse in due parti separate dove A e B sono rispettivamente veri.

Questa relazione ternaria cattura la semantica del tensor e del par. Il par è interpretato dualmente: $A \wp B$ è vero in w se, per ogni decomposizione di w in x e y (cioè per ogni x,y con $R(w,x,y)$), almeno uno tra A in x e B in y è vero. La dualità tra tensor e par si riflette perfettamente nella struttura della relazione ternaria.

Per la logica di rilevanza, la relazione ternaria ha un'interpretazione diversa ma altrettanto significativa. $R(w,x,y)$ significa che il mondo w può testimoniare che dal mondo x è possibile raggiungere il mondo y attraverso un'inferenza rilevante. L'implicazione $A \rightarrow B$ è vera in w se, ogni volta che $R(w,x,y)$ e A è vero in x , allora B deve essere vero in y . Questo garantisce che ci sia una connessione genuina, testimoniata da w , tra le situazioni dove A è vero e quelle dove B è vero.

Le proprietà della relazione R determinano quali principi logici sono validi. Se R è riflessiva nel senso che $R(w,w,w)$ per ogni w , allora l'identità $A \rightarrow A$ è valida: ogni mondo testimonia che da se stesso può raggiungere se stesso. Se R soddisfa una forma di transitività, allora certe forme di composizione di implicazioni diventano valide. Variando le proprietà di R , otteniamo diversi sistemi di logica di rilevanza: R , T , E , e così via.

Semantica a Giochi: Dimostrazioni come Strategie

Una delle interpretazioni più affascinanti e moderne delle logiche substructurali è la game semantics, dove le formule sono viste come giochi tra due giocatori e le dimostrazioni come strategie vincenti. Questa prospettiva non è solo una curiosità matematica: ha portato a risultati profondi sulla semantica dei linguaggi di programmazione e ha ispirato nuovi approcci alla verifica automatica.

Immaginiamo due giocatori: il Proponente (P) cerca di dimostrare che una formula è vera, l'Opponente (O) cerca di falsificarla o almeno di mettere alla prova la dimostrazione. Un gioco consiste in una serie di mosse alternate tra i due giocatori. Ogni mossa corrisponde a una decomposizione logica della formula in gioco.

Consideriamo il tensor $A \otimes B$ nella logica lineare. Il Proponente afferma di poter fornire sia A che B . L'Opponente può sfidarla su entrambi: "Mostrami A " e "Mostrami B ". Il Proponente deve essere in grado di rispondere a entrambe le sfide per vincere. Crucialmente, le risorse disponibili al Proponente devono essere divise tra le due sottopartite: quelle usate per dimostrare A non sono disponibili per dimostrare B .

Il par $A \wp B$ funziona al contrario: l'Opponente decide se sfidare su A o su B, e il Proponente deve essere preparata a rispondere a entrambe le possibilità ma usando tutte le risorse disponibili. Questa asimmetria tra chi sceglie e chi risponde cattura perfettamente la dualità tra tensor e par.

I connettivi additivi coinvolgono scelte vere e proprie. Il with $A \& B$ significa che il Proponente offre una scelta all'Opponente: "Vuoi che dimostri A o che dimostri B?". L'Opponente sceglie, e il Proponente deve essere preparata per entrambe le eventualità, ma dovrà dimostrare solo quella scelta. Il plus $A \oplus B$ inverte i ruoli: il Proponente sceglie quale dimostrare, e l'Opponente deve accettare la scelta.

L'implicazione lineare $A \multimap B$ è interpretata come un gioco dove l'Opponente fornisce un A e il Proponente deve rispondere con un B. Le mosse dell'Opponente nel sottogioco per A diventano mosse del Proponente viste dal gioco complessivo (i ruoli si invertono), mentre le mosse del Proponente nel sottogioco per B restano mosse del Proponente.

Gli esponenziali introducono la possibilità di ripetizione. Se ho $!A$, significa che l'Opponente può chiedere A un numero arbitrario di volte, e il Proponente deve essere preparata a fornirlo ogni volta. Questo corrisponde esattamente alla possibilità di contrazione: posso usare la stessa risorsa più volte.

Una strategia vincente per il Proponente è una funzione che assegna una mossa appropriata del Proponente a ogni possibile sequenza di mosse che termina con una mossa dell'Opponente. Una tale strategia corrisponde esattamente a una dimostrazione nella logica lineare. La composizione di dimostrazioni (la regola del Cut) corrisponde alla composizione di strategie: si giocano due giochi in parallelo, nascondendo le mosse intermedie sulla formula di cut.

La game semantics è stata usata per ottenere risultati di full abstraction per linguaggi di programmazione. Due programmi sono semanticamente equivalenti se e solo se le loro strategie associate sono equivalenti. Questo risolve un problema che era rimasto aperto per vent'anni riguardo al linguaggio PCF (Programming Language for Computable Functions). La logica lineare e la sua game semantics forniscono esattamente il livello di granularità necessario per distinguere tra programmi computazionalmente diversi ma estensionalmente equivalenti.

Semantica Categorica: Quando l'Algebra Incontra la Geometria

La teoria delle categorie fornisce forse la semantica più elegante e generale per le logiche substructurali. Una categoria è una struttura matematica che astrae la nozione di "oggetti e frecce tra loro": gli oggetti possono essere insiemi, spazi, tipi, o qualsiasi cosa abbia senso connettere con morfismi. Le frecce si compongono associativamente, e ogni oggetto ha una freccia identità.

Per interpretare la logica lineare, abbiamo bisogno di una categoria monoidale simmetrica. "Monoidale" significa che c'è un'operazione di prodotto tensoriale sugli oggetti, che corrisponde al connettivo $\otimes$ della logica. "Simmetrica" significa che $A \otimes B$ è isomorfo a $B \otimes A$: l'ordine non conta per il tensor (ma potrebbe contare per altri connettivi in logiche ordinate).

Le formule della logica lineare sono interpretate come oggetti della categoria. I morfismi della categoria rappresentano le dimostrazioni: un morfismo da A a B corrisponde a una dimostrazione di $A \vdash B$. La composizione di morfismi corrisponde alla regola del Cut: se ho una dimostrazione da A a B e una da B a C , la loro composizione è una dimostrazione da A a C .

Per interpretare completamente la logica lineare, abbiamo bisogno di una categoria *-autonoma. Questo significa che esiste una dualità $(\cdot)^\perp$ che inverte le frecce: $A^\perp$ è l'oggetto duale di A , e un morfismo da A a B corrisponde a un morfismo da $B^\perp$ a $A^\perp$. Questa dualità cattura esattamente la negazione lineare.

Il tensor $\otimes$ è il prodotto monoidale. Il par $\wp$ è definito come $A \wp B = (A^\perp \otimes B^\perp)^\perp$: il duale del tensor dei duali. L'implicazione lineare è $A \multimap B = A^\perp \wp B$: il par del duale di A con B . Tutti i connettivi moltiplicativi emergono naturalmente dalla struttura categorica.

I connettivi additivi richiedono che la categoria abbia prodotti e coprodotti finiti. Il with $\&$ è interpretato come il prodotto categorico, il plus $\oplus$ come il coprodotto. Questi sono universali nel senso della teoria delle categorie: il prodotto $A \& B$ è l'oggetto terminale nel diagramma delle proiezioni verso A e B .

Gli esponenziali $!$ e $?$ sono interpretati come funtori: funzioni che portano oggetti in oggetti e morfismi in morfismi, preservando la struttura categorica. Più precisamente, $!$ è una comonade: un funtore con trasformazioni naturali che soddisfano certe equazioni. La comonade $!$ permette di duplicare (contrazione) e scartare (indebolimento) l'oggetto, recuperando esattamente le regole strutturali che la logica lineare pura aveva rigettato.

Questa interpretazione categorica non è solo una curiosità matematica. Ha portato a scoperte profonde sui legami tra logica, teoria della computazione e topologia. La corrispondenza Curry-Howard-Lambek connette tre mondi apparentemente distinti: le dimostrazioni logiche corrispondono ai programmi che corrispondono ai morfismi categorici. Questa triplice corrispondenza è una delle unificazioni concettuali più profonde della matematica moderna.

Parte IX: Applicazioni Reali e Impatto Tecnologico

Rust e il Trionfo della Logica Affine

Quando nel 2010 Graydon Hoare iniziò a sviluppare Rust alla Mozilla Research, probabilmente non pensava esplicitamente alle logiche substructurali. Eppure il sistema di ownership che avrebbe reso Rust rivoluzionario è una implementazione quasi perfetta della logica affine applicata alla gestione della memoria.

Il problema che Rust cerca di risolvere è antico quanto la programmazione stessa: come gestire la memoria in modo sicuro ed efficiente senza un garbage collector? I linguaggi tradizionali offrono due soluzioni insoddisfacenti. Da un lato abbiamo C e C++, dove il programmatore controlla manualmente ogni allocazione e deallocazione. Questo offre

performance massime ma apre la porta a un'intera categoria di bug devastanti: use-after-free, double-free, dangling pointers, buffer overflows. Dall'altro lato abbiamo linguaggi con garbage collection come Java, Python, JavaScript, dove la memoria è gestita automaticamente a costo di performance imprevedibili e overhead continuo.

Rust propone una terza via, basata su un'intuizione profonda: la maggior parte degli errori di gestione memoria derivano dalla violazione implicita di invarianti di ownership. Se più parti del codice possono modificare lo stesso dato, o se posso usare un puntatore dopo che la memoria è stata liberata, è perché non sto rispettando vincoli logici naturali sull'uso delle risorse.

Il sistema di ownership di Rust codifica questi vincoli come regole del type system. Ogni valore ha un owner unico. Quando l'owner esce dallo scope, il valore viene automaticamente deallocato. Posso trasferire l'ownership (move), ma allora il proprietario originale non può più usare il valore. Questo è esattamente la logica affine: ogni risorsa può essere usata al massimo una volta. Non è lineare pura perché posso decidere di non usare un valore (viene semplicemente droppato quando esce dallo scope), ma è affine perché non posso duplicarlo implicitamente.

Il sistema di borrowing completa il quadro. Posso "prestare" un riferimento a un valore senza trasferire l'ownership, ma con vincoli rigidi: posso avere o un numero arbitrario di riferimenti immutabili, o un singolo riferimento mutabile, ma mai entrambi contemporaneamente. Questo previene data races al tempo di compilazione: se nessuno può modificare mentre altri leggono, e solo uno alla volta può modificare, allora le race conditions sono impossibili per costruzione.

I risultati sono straordinari. Rust offre performance comparabili a C e C++ (spesso identiche, a volte superiori grazie a ottimizzazioni che l'assenza di aliasing permette) ma con garanzie di sicurezza che nessun altro linguaggio systems-programming fornisce. Niente use-after-free, niente double-free, niente data races. E tutto questo verificato staticamente, al tempo di compilazione, senza overhead a runtime.

Il successo di Rust non è solo tecnico ma culturale. Ha dimostrato che le idee della logica substrutturale non sono astrazioni accademiche ma strumenti pratici per risolvere problemi reali. Decine di migliaia di programmatori ora ragionano quotidianamente in termini di ownership, borrowing e lifetimes, spesso senza sapere che stanno applicando principi di logica affine sviluppati da logici matematici decenni fa.

Session Types: Protocolli Sicuri per Costruzione

Immaginiamo di dover implementare un protocollo di comunicazione tra client e server. Il client deve prima inviare una richiesta, poi il server risponde con un acknowledgment, poi il client invia i dati, poi il server risponde con un risultato, e infine la connessione si chiude. Sembra semplice, ma nella pratica i protocolli sono facili da implementare scorrettamente: si dimentica di chiudere la connessione, si mandano messaggi nell'ordine sbagliato, si cerca di leggere quando si dovrebbe scrivere.

Le session types, sviluppate da Kohei Honda e collaboratori negli anni Novanta e profondamente radicate nella logica lineare, offrono una soluzione elegante: codificare il

protocollo nel type system. Un session type descrive non solo che tipo di dati vengono scambiati, ma anche in quale ordine e chi fa cosa.

Consideriamo un protocollo semplice per un bancomat. Il cliente inserisce la carta (invia un account ID), il bancomat chiede il PIN (richiede una stringa), il cliente inserisce il PIN (invia la stringa), il bancomat verifica e risponde (invia un booleano), e se la verifica ha successo, il cliente può scegliere tra prelevare o controllare il saldo. Per il prelievo, il cliente invia un ammontare, il bancomat verifica se ci sono fondi sufficienti e risponde di conseguenza, poi rilascia il denaro o nega l'operazione, e infine la sessione si chiude.

Con i session types, questo intero protocollo diventa un tipo:

ATMSession =

Invio[AccountID] poi

Ricezione[PIN] poi

Invio[Bool] poi

Scelta {

Prelievo: Ricezione[Amount] poi Invio[Result] poi Fine,

Saldo: Invio[Balance] poi Fine

}

Il type checker verifica automaticamente che l'implementazione rispetti il protocollo. Se provo a ricevere quando dovrei inviare, o se dimentico di chiudere la sessione, o se uso il canale dopo averlo chiuso, ottengo un errore di tipo al momento della compilazione.

La connessione con la logica lineare è profonda e non superficiale. Ogni operazione sul canale "consuma" il tipo di sessione, trasformandolo nel tipo della continuazione. $\text{Invio}[T] \text{ poi } S$ è come $T \otimes S$: devo fornire sia il dato T che la continuazione S . $\text{Ricezione}[T] \text{ poi } S$ è come $T \wp S$: ricevo T e poi proseguo con S . La Scelta è esattamente il plus $\oplus$: il client decide quale ramo seguire. Se il protocollo prevede che il server scelga, usiamo il with $\&$.

I session types garantiscono tre proprietà fondamentali: communication safety (i tipi dei messaggi corrispondono sempre), session fidelity (il protocollo è seguito correttamente), e deadlock freedom (se tutti seguono i tipi, non ci sono deadlock). Quest'ultima è particolarmente notevole: è un teorema, non una speranza. Se il sistema di tipi accetta il codice, i deadlock sono matematicamente impossibili.

Le applicazioni pratiche stanno emergendo. Il linguaggio Scribble permette di specificare protocolli complessi multi-party e generare automaticamente codice in Java o Python con session types. Il linguaggio Links incorpora session types nativamente per applicazioni web. La libreria session-types per Haskell permette di usare questi concetti in un linguaggio

mainstream. E tool come Singularity OS di Microsoft hanno esplorato sistemi operativi interamente basati su session types per comunicazione tra processi.

Verifica Formale: Separation Logic e Concurrent Separation Logic

Quando John Reynolds e Peter O'Hearn svilupparono la separation logic all'inizio degli anni 2000, risolsero un problema che aveva afflitto la verifica formale per decenni: come ragionare modularmente sulla memoria mutabile.

La logica di Hoare classica, sviluppata negli anni Settanta, permetteva di specificare pre-condizioni e post-condizioni per programmi imperativi. Ma quando la memoria è condivisa e mutabile, il ragionamento diventa intrattabile: modificare una locazione può invalidare assunzioni su altre locazioni se c'è aliasing. La separazione logic introduce il connettivo stella ($P * Q$), la separating conjunction, che afferma "P è vero in una porzione di memoria e Q è vero in una porzione completamente separata".

Questo semplice connettivo, derivato direttamente dalla logica substrutturale, rivoluziona il ragionamento. Se ho $P * Q$, so che P e Q non possono interferire: parlano di regioni di memoria disgiunte. Posso ragionare su di essi indipendentemente. La regola del frame lo rende esplicito: se una funzione richiede preconditione P e garantisce postcondizione Q, e ho anche un altro pezzo di memoria descritto da R che la funzione non tocca, allora posso applicare la funzione a $P * R$ e ottenere $Q * R$. Il pezzo R è "incastonato" intorno all'esecuzione della funzione, garantito immutato.

La concurrent separation logic estende questo al caso multi-thread. Ogni thread "possiede" certe risorse, descritte da una proposizione nella separation logic. Quando un thread vuole accedere a una risorsa posseduta da un altro, deve acquisirla esplicitamente (tramite lock, per esempio), trasferendo l'ownership. Il sistema garantisce che non ci siano data races perché solo il proprietario corrente può modificare una risorsa.

Il framework Iris, sviluppato negli anni 2010 e implementato in Coq, porta queste idee all'estremo. Iris è un framework generale per separation logic concorrente di ordine superiore, con invarianti impredicativi, view shifts, ghost state, e step-indexing per gestire ricorsione. Sembra complesso, e lo è, ma ha permesso di verificare formalmente sistemi che prima erano fuori portata.

Il progetto RustBelt ha usato Iris per verificare la soundness del type system di Rust, dimostrando che il sistema di ownership mantiene davvero le sue promesse. Il progetto ha rivelato alcuni bug sottili nella specifica originale e ha guidato il design di nuove feature del linguaggio. Altri progetti hanno usato Iris per verificare implementazioni di lock-free data structures, runtime di linguaggi con garbage collection, e persino kernel di sistemi operativi.

La lezione profonda è che la logica substrutturale non è solo una teoria ma uno strumento ingegneristico. Fornisce il vocabolario concettuale per esprimere ownership, separazione, trasferimento di risorse, tutti concetti che emergono naturalmente quando si ragiona su sistemi reali. E i tool basati su queste logiche possono verificare proprietà che altre tecniche non possono nemmeno esprimere.

Quantum Computing: No-Cloning e Tipi Lineari

Il computing quantistico porta alla ribalta la logica lineare in modo che nemmeno Girard avrebbe potuto prevedere. I sistemi quantistici seguono leggi che sono naturalmente lineari, non per scelta di design ma per necessità fisica.

Il teorema di no-cloning, dimostrato da Wootters, Zurek e Dieks nei primi anni Ottanta, afferma che è impossibile creare una copia perfetta di uno stato quantistico arbitrario. Formalmente, non esiste un'operazione unitaria U tale che $U(|\psi\rangle \otimes |0\rangle) = |\psi\rangle \otimes |\psi\rangle$ per ogni stato $|\psi\rangle$. Questo è esattamente il rigetto della contrazione: da A non posso derivare $A \otimes A$.

La misurazione di uno stato quantistico lo collassa irrevocabilmente, distruggendolo. Se misuro un qubit in sovrapposizione, ottengo 0 o 1 con certe probabilità, ma lo stato originale è perduto. Questo è il consumo delle risorse nella logica lineare: usare A per ottenere B significa che A non è più disponibile.

I linguaggi di programmazione quantistica devono rispettare queste leggi fisiche. Quipper, sviluppato da ricercatori a Dalhousie e Oxford, usa tipi lineari pervasivamente. Un valore di tipo Qubit può essere usato una sola volta. Se una funzione prende un qubit come input e lo misura, non può restituirlo perché è stato consumato. Se lo applica una porta unitaria, deve restituire il qubit trasformato. Il type checker garantisce che non ci siano duplicazioni implicite o misurazioni premature.

Silq, un linguaggio più recente sviluppato al Politecnico di Zurigo, introduce automatic uncomputation: il compilatore inserisce automaticamente le operazioni per "pulire" i qubit intermedi, restituendoli allo stato zero. Questo è cruciale perché i qubit sono risorse costose (fisicamente!) e devono essere riutilizzati. Ma l'uncomputation deve essere fatta correttamente per preservare la coerenza quantistica. Il sistema di tipi di Silq, basato su logica lineare con estensioni quantitative, garantisce che l'uncomputation sia sicura.

Il ZX-calculus, sviluppato da Coecke e collaboratori, è un linguaggio grafico per ragionare su circuiti quantistici. È essenzialmente una proof net per una logica quantistica, con regole di riscrittura che preservano il significato del circuito. Il ZX-calculus è stato usato per ottimizzare circuiti quantistici, riducendo il numero di porte necessarie, e per verificare l'equivalenza di circuiti apparentemente diversi. La connessione con la logica lineare è diretta: i nodi nel grafo corrispondono a operazioni quantistiche, gli archi a qubit, e le regole di riscrittura a equivalenze logiche che preservano la semantica delle risorse.

Parte X: Grammatica e Linguistica Computazionale

Il Calcolo di Lambek nella Pratica

Quando Joachim Lambek propose il suo calcolo negli anni Cinquanta, aveva in mente un obiettivo ambizioso: fornire una fondazione logica rigorosa per la sintassi del linguaggio naturale. L'idea centrale era che ogni parola o sintagma avesse un tipo logico, e che la grammaticalità di una frase corrispondesse alla derivabilità di un certo sequente in una logica ordinata.

Consideriamo la frase italiana "il gatto nero mangia il topo". Assegniamo tipi:

- "il": un articolo che prende un nome a destra e produce un sintagma nominale: NP/N
- "gatto": un nome: N
- "nero": un aggettivo che prende un nome a sinistra e produce un nome: N\N
- "mangia": un verbo transitivo che prende soggetto a sinistra e oggetto a destra e produce una frase: (NP\S)/NP
- "topo": N

La derivazione procede così. Prima combiniamo "gatto" con "nero": abbiamo N e N\N, che producono N (un gatto nero). Poi combiniamo "il" con "gatto nero": abbiamo NP/N e N, che producono NP (il gatto nero). Similmente per "il topo": NP/N e N producono NP.

Ora abbiamo "il gatto nero" (NP), "mangia" ((NP\S)/NP), e "il topo" (NP). Combiniamo "mangia il topo": (NP\S)/NP e NP producono NP\S. Infine combiniamo "il gatto nero mangia il topo": NP e NP\S producono S, una frase completa.

Se provassimo una frase agrammaticale come "mangia il il gatto topo nero", la derivazione fallirebbe. Non esiste modo di combinare i tipi in questo ordine per produrre S. La grammaticalità emerge dalla derivabilità logica.

Questa idea si è rivelata incredibilmente fertile. Le grammatiche categoriali, basate sul calcolo di Lambek e sue estensioni, sono state usate per parsare lingue naturali, costruire sistemi di question-answering, e persino generare dimostrazioni matematiche formalizzate partendo da testo in linguaggio naturale.

Le grammatiche combinatorie categoriali (CCG), sviluppate da Mark Steedman, estendono Lambek con operatori aggiuntivi che permettono maggiore flessibilità. Le CCG sono state usate con successo in sistemi pratici come il C&C parser, che può analizzare milioni di frasi reali da corpora, e nel sistema di comprensione del linguaggio naturale di IBM Watson.

La connessione profonda è che la sintassi non è arbitraria ma ha una logica interna. L'ordine delle parole, le dipendenze tra sintagmi, le restrizioni su cosa può combinarsi con cosa, tutto questo può essere catturato da un sistema logico dove l'ordine conta e le risorse (le parole) devono essere usate esattamente una volta nel posto giusto.

Discourse Representation Theory e Dinamiche Lineari

La semantica del discorso presenta sfide che vanno oltre la singola frase. I pronomi si riferiscono ad antecedenti introdotti precedentemente, le presupposizioni devono essere soddisfatte dal contesto, e l'ordine delle informazioni influenza l'interpretazione.

Consideriamo: "Un uomo entrò. Lui era alto." Il pronome "lui" si riferisce all'uomo introdotto nella frase precedente. Ma come modelliamo questo formalmente? La Discourse Representation Theory (DRT) di Hans Kamp propone che ogni frase aggiorni una struttura di rappresentazione del discorso (DRS), introducendo referenti e predicati.

La DRT ha un sapore lineare: introdurre un referente lo rende disponibile per pronomi successivi, ma in modo controllato. Non è che il referente viene "copiato" infinite volte;

piuttosto, esiste un unico referente nel contesto discourse che può essere riutilizzato. Questo assomiglia più alla logica affine con possibilità di weakening (posso ignorare referenti introdotti ma non usati) che alla logica lineare pura.

Le semantiche dinamiche, sviluppate da Groenendijk e Stokhof, rendono questa intuizione esplicita: il significato di una frase non è un valore di verità ma una funzione che trasforma contesti. Questo è esattamente l'interpretazione computazionale delle implicazioni in logica lineare: $A \multimap B$ trasforma un contesto dove A è disponibile in un contesto dove B è disponibile, consumando A nel processo.

Le presupposizioni si comportano in modo ancora più interessante. Quando dico "il re di Francia è calvo", presuppongo che esista un re di Francia. Questa presupposizione deve essere soddisfatta dal contesto perché la frase abbia senso. Se il contesto non fornisce un re di Francia, o la presupposizione deve essere accomodata (assumiamo che esista), o la frase è infelice.

Le presupposizioni possono essere proiettate attraverso connettivi in modi complessi. "Se il re di Francia è calvo, allora è umano" presuppone ancora che esista un re di Francia, anche se l'intero condizionale potrebbe essere vero vacuamente. Le logiche substructurali, con la loro attenzione a come le risorse fluiscono attraverso le derivazioni, forniscono strumenti naturali per modellare queste proiezioni.

Parte XI: Filosofia e Fondamenti

Ontologia delle Risorse: Cosa Rende Qualcosa una Risorsa?

La distinzione tra risorse e non-risorse non è così ovvia come sembra. Il denaro è chiaramente una risorsa: è scarso, consumabile, esclusivo. Ma che dire della conoscenza? Se ti insegno un teorema matematico, io non lo perdo. Possiamo entrambi conoscerlo. Eppure acquisire conoscenza richiede risorse: tempo, attenzione, energia cognitiva.

La logica classica modella bene le verità matematiche eterne: "due più due fa quattro" è vero, e resta vero indipendentemente da quante volte lo usiamo in un ragionamento. Non si consuma, non si esaurisce. Ma la logica classica cattura male il processo del ragionamento stesso, che è invece consumante: ho un budget limitato di attenzione, di memoria di lavoro, di tempo.

Le logiche substructurali permettono di distinguere tra il contenuto di una proposizione e le risorse necessarie per stabilirla o usarla. Posso avere $!A$, accesso illimitato alla verità A , pur avendo solo risorse limitate per verificarla o applicarla. Questa distinzione è filosoficamente profonda: separa l'ontologia (cosa esiste) dall'epistemologia (come lo conosciamo) e dalla pragmatica (come lo usiamo).

Il tempo emerge come la risorsa fondamentale. È scarso (abbiamo tutti 24 ore al giorno), consumabile (il tempo passato non torna), ed esclusivo (un istante può contenere solo

un'azione). Le logiche temporali lineari catturano questa intuizione: il tempo fluisce in una direzione, gli eventi si consumano mentre accadono, e l'ordine conta.

Anche l'informazione quantistica risulta essere una risorsa in senso profondo. Il teorema di no-cloning non è solo una limitazione tecnica ma una legge fondamentale della natura. L'informazione quantistica non può essere duplicata, e la misurazione la distrugge irrevocabilmente. È la natura stessa a imporre la linearità, non una scelta di modellazione.

Epistemologia del Ragionamento: I Limiti Cognitivi Reali

Gli esseri umani reali, e i computer reali, hanno limiti cognitivi e computazionali che la logica classica ignora completamente. Abbiamo memoria finita, tempo limitato, capacità di attenzione ristretta. Quando ragioniamo, non possiamo esplorare tutte le possibili derivazioni: dobbiamo fare scelte, seguire euristiche, talvolta backtracking quando ci accorgiamo di aver preso una strada sbagliata.

La logica classica assume risorse infinite per il ragionamento. Posso duplicare premesse infinite volte (contrazione illimitata), posso ignorare premesse irrilevanti senza costo (indebolimento gratuito), posso esplorare ogni possibile riordinamento (scambio libero). Ma nella realtà, ogni operazione ha un costo. Ogni duplicazione richiede memoria, ogni esplorazione richiede tempo, ogni backtracking richiede energia.

Le logiche substrutturali permettono di modellare esplicitamente questi costi. Le annotazioni quantitative in logica lineare quantitativa possono rappresentare quante copie di una risorsa sono disponibili, o quanto tempo è necessario per una computazione, o quanta memoria occupa una struttura dati. Il type system diventa non solo un verificatore di correttezza ma un controllore di risorse.

Questo ha implicazioni profonde per l'intelligenza artificiale. Un agente razionale ideale nella tradizione bayesiana aggiorna le sue credenze usando il teorema di Bayes, considera tutte le possibili azioni, e sceglie quella che massimizza l'utilità attesa. Ma questo richiede risorse computazionali illimitate. Un agente reale deve operare sotto bounded rationality: ragionare bene abbastanza, abbastanza velocemente, con le risorse disponibili.

Le logiche substrutturali forniscono un framework formale per bounded rationality. Posso specificare esplicitamente: "Ho 10 millisecondi per decidere. Ho 1 megabyte di memoria disponibile. Quali conclusioni posso ragionevolmente inferire date queste risorse?" La risposta non è "tutte le conseguenze logiche" ma "quelle derivabili dentro i vincoli di risorse specificati".

Rilevanza e Significato: Quando un'Inferenza È Genuina?

La questione filosofica centrale della logica di rilevanza è: quando un'implicazione è genuina? La logica classica dice: A implica B quando è impossibile che A sia vero e B falso. Ma questo criterio puramente estensionale ignora completamente se c'è una connessione di significato tra A e B.

"Se la Luna è di formaggio, allora Parigi è in Francia" è vera nella logica classica (la premessa è falsa, quindi l'implicazione è vacuamente vera). Ma cattura davvero

un'implicazione? C'è qualche senso in cui la composizione della Luna è rilevante per la geografia terrestre? Intuitivamente no, e la logica di rilevanza formalizza questa intuizione.

Ma cosa significa esattamente "rilevanza"? Anderson e Belnap proposero il criterio di uso variabile: A è rilevante per B in una derivazione se qualche variabile proposizionale appare sia in A che in B, e quella variabile è effettivamente usata nella derivazione. Questo cattura l'idea che ci deve essere un filo conduttore concettuale che connette premessa e conclusione.

Un criterio alternativo, più semantico, è quello di connessione causale o esplicativa: A è rilevante per B se A fornisce una ragione, una causa, o una spiegazione per B. "Se piove, la strada è bagnata" è rilevante perché la pioggia causa o spiega il bagnato della strada. "Se due più due fa quattro, la strada è bagnata" non è rilevante perché non c'è connessione causale o esplicativa, anche se l'implicazione è materialmente vera (la premessa è vera).

La rilevanza ha implicazioni per la filosofia della scienza. Le leggi scientifiche dovrebbero essere rilevanti nel senso tecnico: dovrebbero connettere genuinamente causa ed effetto, non essere mere correlazioni accidentali. Le spiegazioni scientifiche dovrebbero essere rilevanti: dovrebbero mostrare perché qualcosa accade, non solo elencare fatti veri che accadono simultaneamente.

La Negazione come Dualità: Una Rivoluzione Concettuale

Nella logica classica, la negazione è primitiva e asimmetrica: $\neg A$ significa "A è falso". Ma nella logica lineare, la negazione diventa simmetrica e derivata: $A^\perp$ è il duale di A, ciò che interagisce con A per produrre neutralità.

Questa prospettiva cambia radicalmente il modo di pensare alla negazione. Non è più questione di verità e falsità, ma di domanda e offerta, di consumo e produzione. Se A rappresenta "offro una mela", allora $A^\perp$ rappresenta "richiedo una mela". Se fornisco ciò che è richiesto, le risorse si annullano: $A \otimes A^\perp$ produce neutralità (rappresentata dalla costante $\perp$).

La dualità è involutiva: $(A^\perp)^\perp = A$. Questo è sorprendente e non banale. Nella logica intuizionista, la doppia negazione non è equivalente all'originale: $\neg\neg A$ non implica A in generale. Ma nella logica lineare, negare due volte ritorna esattamente all'inizio. Questo riflette una simmetria profonda nella struttura: non c'è asimmetria tra ciò che offro e ciò che richiedo, tra input e output, tra risorsa consumata e risorsa prodotta.

Filosoficamente, questo suggerisce che la negazione non è una primitiva ontologica ma emerge dalla struttura delle interazioni. Non esiste "il falso" come entità indipendente; esiste solo il fallimento dell'interazione, l'impossibilità di soddisfare simultaneamente domande incompatibili. La verità e la falsità non sono proprietà intrinseche delle proposizioni ma relazioni tra proposizioni e contesti di valutazione.

Determinismo e Libero Arbitrio: Implicazioni Metafisiche

La logica lineare ha implicazioni sottili per il dibattito sul determinismo. Da un lato, ogni uso di una risorsa è tracciabile: dato lo stato iniziale delle risorse, l'evoluzione del sistema è

determinata. Questo rispecchia il determinismo fisico: date le leggi della natura e lo stato iniziale dell'universo, tutto ciò che segue è fissato.

Ma dall'altro lato, i connettivi additivi $\oplus$ e $\&$ rappresentano scelte genuine. Il plus $A \oplus B$ significa che avverrà A oppure B, ma la scelta non è determinata dal sistema: è esterna, fornita dall'ambiente. Il with $A \& B$ significa che posso scegliere se procedere con A o con B. Queste scelte introducono una forma di indeterminismo nella logica.

Nel quantum computing, questa distinzione diventa fisica. L'evoluzione unitaria di uno stato quantistico è completamente deterministica: la equazione di Schrödinger è deterministica. Ma la misurazione introduce genuino indeterminismo: dato uno stato in sovrapposizione, quale outcome otterremo è genuinamente casuale (secondo l'interpretazione ortodossa). La logica lineare quantistica distingue tra operazioni deterministiche (tensor, implicazione) e operazioni indeterministiche (misurazione, rappresentata dai connettivi additivi).

Per il libero arbitrio, questo suggerisce una posizione intermedia tra determinismo assoluto e libertarismo radicale. Le scelte avvengono dentro vincoli di risorse: non posso scegliere arbitrariamente senza limiti, ma dentro i vincoli posso fare scelte genuine che non sono determinate dal passato. Il with $\&$ rappresenta questo tipo di scelta compatibilista: date le risorse disponibili, scelgo come usarle.

Parte XII: Direzioni di Ricerca Contemporanea

Dependent Linear Types: Unificare Dipendenza e Linearità

Una delle frontiere più eccitanti della ricerca attuale è l'integrazione tra tipi dipendenti e tipi lineari. I tipi dipendenti, centrali in Coq, Agda, Idris, permettono che i tipi dipendano da valori: posso avere il tipo "vettore di lunghezza n" dove n è un numero naturale. Questo permette di esprimere specifiche incredibilmente precise: non solo "questa funzione prende una lista", ma "questa funzione prende una lista di lunghezza n e restituisce una lista di lunghezza n+1".

I tipi lineari, come abbiamo visto, permettono di controllare le risorse: quante volte un valore è usato, se può essere duplicato o scartato. L'integrazione dei due sistemi permetterebbe di esprimere specifiche come: "questa funzione prende un buffer di n bytes, lo usa esattamente una volta, e restituisce un buffer di n bytes più una checksum di 4 bytes, consumando il buffer originale".

Ma integrare i due sistemi è tecnicamente difficilissimo. Il problema sta nell'interazione tra eliminazione di dipendenze e linearità. Nei tipi dipendenti standard, posso eliminare un tipo dipendente da un valore anche se poi non uso quel valore nella computazione: il valore esiste a livello di tipi, non necessariamente a runtime. Ma nei tipi lineari, ogni valore deve essere usato esattamente una volta. Come riconciliare queste due esigenze?

Le proposte attuali includono Quantitative Type Theory (QTT) di Atkey, che annota ogni binding con una quantità (0, 1, o ω) indicando quante volte deve essere usato. Un binding

con quantità 0 esiste solo a livello di tipi, un binding con quantità 1 è lineare, un binding con quantità ω è classico. Questo permette di mescolare i tre regimi nello stesso linguaggio.

Un'altra proposta è Linear Dependent Type Theory di Krishnaswami e collaboratori, che distingue esplicitamente tra contesto computazionale (lineare) e contesto di tipi (non-lineare). I termini nel contesto di tipi possono essere usati liberamente per formare tipi dipendenti, ma non sono computazionalmente rilevanti. I termini nel contesto computazionale devono essere usati linearmente.

Le applicazioni potenziali sono enormi. Potremmo verificare formalmente protocolli crittografici garantendo che le chiavi segrete non vengano accidentalmente duplicate o esposte. Potremmo implementare linguaggi per quantum computing dove il type system garantisce sia la linearità fisica dei qubit sia proprietà dipendenti come "questo circuito ha esattamente n porte CNOT". Potremmo costruire sistemi operativi dove il type system garantisce che le risorse hardware (memoria, file handles, interrupts) siano gestite correttamente e che i driver soddisfino specifiche dipendenti precise.

Homotopy Type Theory e Logiche Higher-Dimensional

L'Homotopy Type Theory (HoTT), sviluppato da Voevodsky e collaboratori, interpreta i tipi come spazi topologici e i termini come punti in quegli spazi. Le uguaglianze diventano path: due termini sono uguali se esiste un path che li connette. Questa prospettiva ha rivoluzionato i fondamenti della matematica, fornendo un'alternativa alla teoria degli insiemi come fondazione.

Ma come si integra HoTT con le logiche substructurali? Questa è una domanda aperta affascinante. Potremmo avere una logica lineare omotopica dove le risorse sono spazi topologici e le trasformazioni sono omotopie? Cosa significherebbe che due risorse sono "omotopicamente equivalenti"?

Alcuni primi passi sono stati fatti. La logical relations in HoTT possono essere viste come generalizzazioni delle relazioni nella semantica categorica della logica lineare. Le higher inductive types in HoTT permettono di definire tipi con path e higher paths specificati, similmente a come i connettivi nella logica lineare sono definiti tramite regole di introduzione ed eliminazione con proprietà di dualità.

Una direzione promettente è l'interpretazione delle proof nets come diagrammi di stringhe in categorie higher. Le proof nets della logica lineare sono grafi planari; potremmo generalizzare a complessi simpliciali higher-dimensional dove le facce rappresentano equivalenze omotopiche tra derivazioni. Questo permetterebbe di ragionare non solo su cosa è derivabile ma su come derivazioni diverse sono equivalenti modulo deformazioni continue.

Machine Learning e Differential Programming

La connessione tra logica lineare e machine learning, mediata dalla Differential Linear Logic (DiLL) di Ehrhard e Regnier, è una delle scoperte più sorprendenti degli ultimi anni. Il calcolo differenziale, fondamento del training delle reti neurali via backpropagation, può essere interpretato logicamente.

L'idea chiave è introdurre un operatore di derivata nella logica lineare: dato un termine di tipo $A \multimap B$, rappresentante una funzione da A a B , possiamo costruire un termine di tipo $A \otimes \partial A \multimap B \otimes \partial B$ che rappresenta la funzione insieme alla sua derivata. Qui ∂A rappresenta "lo spazio tangente" al tipo A , e $\partial A \rightarrow B$ rappresenta la linearizzazione della funzione attorno a un punto.

Questa costruzione cattura esattamente la tecnica dei dual numbers usata nell'automatic differentiation: se $x + \epsilon dx$ rappresenta un numero con la sua derivata infinitesimale, allora $f(x + \epsilon dx) = f(x) + \epsilon f'(x)dx$ calcola simultaneamente il valore e la derivata. Il tipo ∂A formalizza l' ϵ .

Ma DiLL non si ferma qui. Permette di comporre derivate in modi complessi, gestire derivate di ordine superiore, e ragionare sulla correttezza di algoritmi di ottimizzazione. La regola di chain rule del calcolo diventa una regola di composizione nella logica: se ho $f: A \multimap B$ e $g: B \multimap C$, la derivata di $g \circ f$ è data dalla composizione delle derivate di f e g .

I linguaggi di differentiable programming come JAX, Dex, e i framework come PyTorch sono ispirati (spesso inconsapevolmente) da queste idee. Definire una rete neurale è essenzialmente costruire un termine in una logica lineare differenziale. Il training via backpropagation è cut-elimination con derivate. Le garanzie di correttezza dei compilatori di questi linguaggi potrebbero essere formalmente verificate usando DiLL come semantica.

Guardando al futuro, potremmo immaginare linguaggi dove il type system garantisce non solo la correttezza del programma ma anche proprietà delle sue derivate: convessità, lipschitz-continuità, stabilità numerica. Un compilatore potrebbe ottimizzare non solo il codice ma anche il gradiente, trovando modi più efficienti di calcolare backpropagation. La verifica formale potrebbe estendersi a garantire che una rete neurale trained soddisfi certe proprietà di robustezza o fairness, codificate come tipi dipendenti lineari differenziali.

Logiche substructurali per Blockchain e Smart Contracts

I sistemi blockchain e gli smart contracts presentano sfide uniche per la correttezza e la sicurezza. Gli asset digitali su blockchain sono intrinsecamente lineari: un token non può essere duplicato, non può essere creato dal nulla, non può essere distrutto arbitrariamente. Ogni trasferimento consuma il token dal mittente e lo produce nel destinatario. Questo è esattamente la logica lineare.

Il linguaggio Move, sviluppato per la blockchain Diem (ex Libra), tratta gli asset come resources nel senso tecnico delle logiche substructurali. Un resource type non può essere copiato o scartato implicitamente. Deve essere esplicitamente creato (da codice autorizzato), trasferito (consumandolo dal sender e producendolo nel receiver), o distrutto (da codice autorizzato). Il type system garantisce che non ci sia double-spending: non posso usare lo stesso asset due volte.

Obsidian, sviluppato alla CMU, aggiunge tpestate alla logica lineare per smart contracts. Uno smart contract può essere in diversi stati (per esempio, un'asta può essere Open, Closed, o Finalized), e solo certe operazioni sono permesse in certi stati. Il type system traccia lo stato corrente e previene transizioni illegali. Combina questo con ownership lineare: solo il proprietario corrente di un asset può operare su di esso.

Il problema del reentrancy, che causò il famigerato DAO hack su Ethereum (perdita di \$50 milioni), è impossibile nei sistemi con tipi lineari appropriati. Il reentrancy bug accade quando un contratto chiama una funzione esterna che callback nel contratto originale prima che la prima chiamata finisca, creando stati inconsistenti. Ma se l'ownership è lineare, quando trasferisco controllo a un contratto esterno, perdo temporaneamente l'ownership del mio stato. Il contratto esterno non può callback perché non ha l'ownership necessaria. Quando il controllo ritorna, riacquisto l'ownership, ma a quel punto la transazione originale è completata.

Scilla, il linguaggio per la blockchain Zilliqa, separa esplicitamente computazione (funzioni pure) e comunicazione (message passing con ownership transfer). Le funzioni pure non possono avere side effects e sono verificate usando logical relations. Il message passing è lineare: ogni messaggio trasferisce ownership delle risorse. Il risultato è che Zilliqa ha avuto zero vulnerabilità critiche in produzione da quando ha adottato Scilla.

Parte XIII: Prospettive Future e Grandi Questioni Aperte

La Grande Unificazione: Esiste una Teoria di Tutto per le Logiche?

Una delle grandi domande che affascina i ricercatori è: esiste una teoria unificata che comprende tutte le logiche substructurali, i tipi dipendenti, l'homotopy type theory, le modalità, e la computazione quantistica? O queste sono famiglie genuinamente separate di idee che si sovrappongono in modi interessanti ma non possono essere completamente unificate?

Alcuni ricercatori, ispirati dallo slogan di Girard "la logica lineare è la logica dietro la logica", credono che esista un framework più generale dove la logica classica, intuizionista, lineare, rilevante, e quantistica sono tutte proiezioni o frammenti di una struttura più ricca. Questo framework includerebbe:

Una teoria dei tipi con quantità generalizzate, dove ogni variabile è annotata con un elemento di un semiring qualsiasi. Il semiring $\{0,1\}$ dà tipi affini, il semiring $\{0,1,\omega\}$ dà logica lineare con esponenziali, il semiring $[0,1]$ dà logica probabilistica, il semiring dei reali positivi dà differential programming, e così via.

Modalità strutturate in un reticolo, dove ogni modalità ha regole strutturali associate. Salendo nel reticolo aggiungo regole strutturali (più permissivo), scendendo le tolgo (più restrittivo). Le logiche diverse corrispondono a scegliere diversi punti o sottoreticoli in questo spazio.

Categorie enriched in higher categories, dove le morphism-sets sono sostituiti da spazi topologici o categorie, permettendo di catturare simultaneamente la struttura categorica della logica lineare e la struttura omotopica di HoTT.

Teoria dei giochi generalizzata dove i giochi possono essere quantistici, probabilistici, o deterministici, e le strategie corrispondono a dimostrazioni in diverse logiche a seconda della struttura del gioco.

Ma forse questa ricerca di unificazione è chimerica. Forse diverse logiche sono genuinamente adatte a diversi domini, e forzare un framework unico sarebbe perdere le distinzioni sottili che rendono ogni logica adatta al suo compito. La matematica non ha bisogno di essere unificata per essere bella o utile.

Decidibilità e Complessità: Quanto Difficile È Ragionare?

Per molte logiche substructurali, la complessità computazionale del problema di derivabilità è ben compresa. Ma ci sono ancora questioni aperte importanti:

Qual è la complessità esatta del sistema R di logica di rilevanza? Sappiamo che è in PSPACE, ma è PSPACE-completo? Esistono frammenti più efficienti che catturano ancora l'essenza della rilevanza?

Per la logica lineare del primo ordine con quantificatori limitati, qual è il confine tra decidibilità e indecidibilità? Il frammento $\exists \forall$ (esiste universale) è indecidibile, ma esistono frammenti più grandi ancora decidibili?

Esistono logiche substructurali che catturano esattamente classi di complessità diverse da P (catturato da Light Linear Logic)? Potremmo caratterizzare L (logarithmic space), o NL (nondeterministic logarithmic space), o le classi della gerarchia polinomiale tramite restrizioni logiche?

Per le logiche ordinate non-associative, quanto diventa difficile il parsing quando aggiungiamo feature linguistiche realistiche come movimenti a lunga distanza, coordinazione, ellissi? Il calcolo base è NP-completo, ma i frammenti utili per linguaggio naturale potrebbero essere più trattabili.

Queste non sono domande accademiche. La complessità determina quali problemi possiamo risolvere in pratica. Se una logica è PSPACE-completa, verificare dimostrazioni è fattibile per problemi piccoli ma intangibile per problemi grandi. Se è NP-completa, possiamo usare SAT solvers moderni e tecniche di approssimazione. Se è in P, possiamo scalare a problemi enormi.

Applicazioni Non Ancora Esplorate: Dove Andranno le Logiche Substructurali?

Guardando al futuro, dove potrebbero andare queste idee? Alcune speculazioni:

Biologia sintetica: Progettare circuiti genetici è essenzialmente assemblare componenti che devono attivarsi in ordini specifici con risorse limitate (molecole di ATP, ribosomi, ecc.). Le logiche ordinate lineari potrebbero fornire un linguaggio di specifica dove la correttezza è verificabile automaticamente. Un compilatore potrebbe tradurre specifiche logiche in sequenze di DNA effettive.

Economia e meccanismi di mercato: I mercati sono sistemi dove agenti scambiano risorse seguendo protocolli. Gli smart contracts su blockchain sono implementazioni esplicite di meccanismi economici. Le logiche substructurali potrebbero formalizzare proprietà

desiderabili come incentive compatibility, strategy-proofness, individual rationality, e verificarle automaticamente nel design del meccanismo.

Neuroscienze computazionali: Il cervello opera sotto vincoli metabolici stringenti. I neuroni non possono sparare arbitrariamente: consumano ATP, hanno periodi refrattari, competono per risorse. Le reti neurali artificiali ignorano questi vincoli, ma le reti neurali biologiche no. Modelli basati su logiche substrutturali potrebbero catturare meglio le dinamiche biologiche reali, portando a architetture più efficienti ed energy-aware.

Linguistica cognitiva: Come rappresentiamo la sintassi nel cervello? Le grammatiche categoriali basate su logiche ordinate potrebbero non essere solo modelli formali ma riflettere reali meccanismi neuronali. Esperimenti di neuroimaging potrebbero testare se l'attività cerebrale durante il processing linguistico corrisponde alle derivazioni in calcolo di Lambek.

Sistemi autonomi multi-agent: Robot che coordinano per compiti complessi devono comunicare, negoziare risorse, garantire assenza di deadlock. Session types e logiche lineari concorrenti potrebbero essere il fondamento formale per protocolli di coordinamento verificabilmente corretti, garantendo che flotte di droni o rover marziani non entrino in stalli.

Crittografia e privacy: I protocolli crittografici spesso richiedono che certe informazioni siano usate esattamente una volta (chiavi temporanee) o mai (in zero-knowledge proofs). Tipi lineari con ownership potrebbero garantire queste proprietà staticamente. Le privacy-preserving computations potrebbero essere espresse come derivazioni in logiche dove l'informazione sensibile è linearmente controllata.

Conclusione: La Logica del Mondo Reale

Siamo partiti dall'osservazione che la logica classica assume regole strutturali invisibili: possiamo duplicare, scartare, e riordinare premesse liberamente. Queste assunzioni sono naturali quando ragioniamo su verità matematiche eterne, ma falliscono quando ragioniamo su risorse concrete, processi computazionali, protocolli di comunicazione, o sistemi fisici.

Le logiche substrutturali rigettano sistematicamente una o più di queste regole, rivelando un universo sorprendentemente ricco di sistemi formali. La logica lineare rigetta contrazione e indebolimento, modellando risorse che si consumano esattamente. La logica affine ammette lo scarto ma non la duplicazione, perfetta per ownership semantics. La logica di rilevanza rigetta l'indebolimento, richiedendo che ogni premessa sia genuinamente usata. Le logiche ordinate rigettano lo scambio, catturando situazioni dove l'ordine conta. Le logiche non-associative vanno oltre, preservando anche la struttura di raggruppamento.

Queste non sono curiosità accademiche. Sono strumenti pratici con applicazioni immediate e importanti:

Rust usa logica affine per memory safety senza garbage collection. Session types garantiscono protocolli di comunicazione corretti e deadlock-free. Separation logic permette

verifica compositiva di programmi concorrenti, usata per verificare il runtime di Rust, kernel di sistemi operativi, e smart contracts. Il calcolo di Lambek fornisce fondamenta per grammatiche categoriali che parsano linguaggio naturale. I linguaggi quantistici usano tipi lineari per prevenire clonage e garantire correttezza quantistica. Differential linear logic ispira linguaggi di differentiable programming per machine learning.

Ma le implicazioni vanno oltre le applicazioni immediate. Le logiche substrutturali cambiano il modo stesso di pensare alla logica e al ragionamento:

Non c'è "la" logica corretta. Diverse situazioni richiedono diverse logiche, con diverse regole strutturali appropriate. La logica è pluralistica, non monolitica.

La distinzione risorse/verità non è assoluta ma dipende dal contesto. Anche la "conoscenza" può essere una risorsa quando consideriamo i costi di acquisirla o trasmetterla.

La negazione non è primitiva ma può emergere come dualità. $A^\perp$ non è "A è falso" ma "ciò che interagisce con A per produrre neutralità".

Le dimostrazioni non sono solo certificati di verità ma oggetti computazionali con semantica operativa. Hanno costo in tempo, memoria, energia.

L'ordine, la struttura, il flusso di risorse sono aspetti fondamentali del ragionamento, non dettagli implementativi da astrarre via.

Guardando al futuro, le logiche substrutturali continueranno a crescere in importanza. Man mano che costruiamo sistemi più complessi—sistemi distribuiti, blockchain, intelligenza artificiale, computer quantistici, biologia sintetica—avremo sempre più bisogno di strumenti formali che catturano vincoli di risorse, protocolli di comunicazione, garanzie di sicurezza. Le logiche classiche e intuizioniste resteranno fondamentali per la matematica pura, ma per l'informatica, l'ingegneria, e le scienze applicate, le logiche substrutturali diventeranno altrettanto centrali.

La logica classica fu la logica dell'eternità, di verità fuori dal tempo. Le logiche substrutturali sono le logiche del tempo, delle risorse, del cambiamento, del costo, del mondo reale. Sono le logiche del XXI secolo.